User Transparent Parallel Image Processing

Seinstra, F.J.

Publication date
2003
Document Version
Final published version

User Transparent Parallel Image Processing

Frank J. Seinstra

User Transparent Parallel Image Processing

Frank J. Seinstra

This book is typeset by the author using LaTeX2$_\varepsilon$. The main body of the text is set using Times font © Adobe Systems Incorporated. The images and figures are included in the text in encapsulated Postscript format ^{TM} Adobe Systems Incorporated.

Printing: Febodruk BV, Enschede, The Netherlands.

The graphic on the cover symbolically represents many of the aspects of the software architecture for user transparent parallel image processing described in this thesis. The image of the roundel, with its blue bar against a bright red circle, hints at the tube station signs used at London Underground. Just as metro trains are speeding throughout an underground network of tunnels, hidden from anyone directly above it, in our software architecture all intricacies of high speed processing are shielded from the user completely. Also, the tube station sign is affected somewhat by motion blur, to indicate the speed obtained with our architecture. The partitioning of the image hints at the strictly data parallel approach followed in all implementations. Finally, the characters in the blue bar spell out: "Mind the gap". First, this refers to the research issue of Chapter 5, which stresses the importance of incorporating memory layout in the modeling of message passing programs. More importantly, this relates to the observed gap between the highly specialized expertise of the image processing community, and the additional expertise required for efficient employment of high performance computer architectures. Only recognition of this particular gap can bring about an acceptable long-term solution for the image processing community at large.

ISBN 90-5776-102-5

User Transparent Parallel Image Processing

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam,
op gezag van de Rector Magnificus prof. mr. P.F. van der Heijden
ten overstaan van een door het College voor Promoties ingestelde commissie,
in het openbaar te verdedigen in de Aula der Universiteit
op donderdag 8 mei 2003 te 14.00 uur

door

Frank Johan Seinstra

geboren te Haarlem

Promotiecommissie:

Promotor:	Prof. dr. ir. A.M.W. Smeulders
Co-promotor:	dr. D.C. Koelma
Overige leden:	Prof. dr. ir. H.E. Bal Prof. dr. L.O. Hertzberger Prof. dr. M.L. Kersten Prof. dr. ir. H.J. Sips dr. ir. P.P. Jonker
Faculteit:	Faculteit der Natuurwetenschappen, Wiskunde & Informatica Kruislaan 403 1098 SJ Amsterdam Nederland

The work described in this thesis was supported by the Netherlands Organisation for Scientific Research (NWO) under grant 612-11-000.

NWO

Advanced School for Computing and Imaging

The work described in this thesis has been carried out within graduate school ASCI, at the Intelligent Sensory Information Systems group of the University of Amsterdam. ASCI dissertation series number 84.

isis

Intelligent Sensory Information Systems
University of Amsterdam
The Netherlands

Contents

Chapter 1

Introduction

"The fact that the pieces do fit together [...] is something you might miss from focussing too closely on one aspect of science."

John Gribbin - *Almost Everyone's Guide to Science* (1998)

Throughout history, mankind has had an ever growing desire for increased efficiency. Irrespective of the origin of the desire, its manifestations are manifold. For example, the desire to efficiently disseminate ideas and information from a single source to a large and far-ranging audience, directly led to Gutenberg's invention of the printing press*. In the Mid-Eighteenth Century, the desire for large-scale production resulted in the application of power-driven machinery to manufacturing — and the Industrial Revolution. More recently, the need for automated processing of scientific problems, and the handling of large amounts of data, led to the advent of the Information Age.

Once its raison d'être is demonstrated, high-speed machinery is constantly being improved upon for ever increased efficiency. A good example is the development of successive generations of trains. When the English inventor Richard Trevithick introduced the steam locomotive on 21 February 1804 in Wales, it achieved a speed of 8 km/h. In 1825, Englishman George Stephenson introduced the world's first workable passenger train, which steamed along at 24 km/h. Today, the fastest passenger trains fly down the tracks at a speed of approximately 550 km/h.

Part of the success of such technologies stems from the fact that successive performance improvements generally did not result in increased user requirements. At any time, a passenger could get onto a train, sit down pleasantly, get some sleep, read a newspaper, do some work, and get off at any station, *without ever having to worry about the actual running of the train*. If, in contrast, it would have been required

*For reasons of completeness, the author would like to stress that in his city of birth (Haarlem, The Netherlands) many still consider Laurens Jansz. Coster to be the true inventor of printing [160].

from anyone travelling the latest zenith in high-speed train design to have expert knowledge regarding the train's locomotion, passenger numbers would have dropped dramatically. Unfortunately, it is exactly this problem that can be observed with respect to the latest developments in high performance computer systems.

As stated, the Information Age has seen major breakthroughs in the automated processing of scientific problems. Today, ever more complex problems are being studied using ever faster, and more complex machines. Often, the required processing power is delivered only by arguably the most complex systems of all — i.e., high performance parallel computers in their myriad of forms. To effectively exploit the available processing power, a thorough understanding of the complexity of such systems is required. As an immediate consequence, the number of 'passengers' that is capable of 'riding' such high performance parallel architectures is low.

Despite the complexity, many non-expert users are still tempted by the processing power provided by parallel systems — often to emerge with nothing but a disappointing result. In [75] this problem is stated somewhat more dramatically as follows:

> Anecdotal reports abound about researchers with scientific and engineering problems who have tried to make use of parallel processing systems, and who have been almost fatally frustrated in the attempt.

Clearly, there is a major discrepancy between the desire to obtain high performance with relative ease, and the potential of current high performance systems (i.e., the combination of all software layers and the underlying hardware) to satisfy this desire.

As indicated below, the specific research area of image processing — which is the field of focus of this thesis — also demonstrates a persistent desire to access the speed potential of high performance computer systems. The desire partially stems from the fact that it has been recognized for years that the application of parallelism in image processing can be highly beneficial [161]. However, in the field of image processing research, the observed discrepancy between desire and reality is no less severe. Essentially, the work described in this thesis is an endeavor to resolve this discrepancy — and to satisfy the need for easily obtainable speed in image processing.

1.1 The Need for Speed in Image Processing

The 'need for speed' has been recognized in many areas of digital image processing and computer vision [151]. Applications abound in which large amounts of data are to be processed, while having to adhere to strict time constraints at the same time. For example, a typical visual information standard such as television may generate data at a rate of up to 120 Mbytes per second [130]. As each pixel in the information stream generally is subjected to a multitude of processing steps, the total amount of processing power required per time unit is huge. In many cases (e.g., when real-time requirements are to be met), state-of-the-art sequential computers no longer can provide the necessary performance. The only way to supply the desired processing power (now and in the future) is by employing high performance computer systems.

A considerable diversity exists in the type of algorithms applied in imaging applications. Generally, a distinction is made between three different operation levels [118]:

1. **Low level image processing operations**. These operations primarily work on whole image data structures, and yield another image data structure. The computations have a local nature, and are to be performed for each pixel in an image. Examples are: basic filter operations (e.g., smoothing, edge enhancement), and image transformations (e.g., rotation, scaling).
2. **Intermediate level image processing operations**. These operations reduce the image data field into segments (regions of interest), and produce more compact and symbolic image structure representations (such as lists of object border descriptions). Examples are: region labeling, and object tracking.
3. **High level image processing operations**. These operations primarily concern the interpretation of the symbolic data structures obtained from the intermediate level operations. Essentially, the operations try to imitate human cognition and decision making, according to the information contained in the image. Examples are: object recognition, and semantic scene interpretation.

The execution of a set of low level routines is a common starting point for many typical image processing applications. In this thesis, we restrict ourselves to this initial processing phase. First, this is because the processing of visual data at the pixel level is highly regular in nature, to the effect that it provides a natural source of parallelism. More importantly, this is because the initial processing phase is by far the most time consuming part of the bulk of image processing applications [165].

1.2 The Gap Between Computing and Imaging

In spite of the large potential performance gains (and the overwhelming desire to obtain them), the image processing community at large does not benefit from high performance computing on a daily basis. As will be discussed extensively in this thesis, the problem is primarily due to the fact that no programming tool is available that can effectively help non-expert parallel programmers in the development of image processing applications for efficient execution on high performance computing architectures. Existing programming tools generally require the user to identify the available parallelism at a level of detail that is beyond the skills of non-expert parallel programmers [148]. As it is unrealistic to expect researchers in the field of image processing to become experts in high performance computing as well, it is essential to provide a tool that shields its users from *all* intrinsic complexities of parallelization.

The work described in this thesis is an attempt to effectively bridge the gap between the specific expertise of the image processing community, and the additional expertise required for efficient employment of high performance computer architectures. More specifically, the thesis describes the design and implementation of a *software architecture* that allows non-expert parallel programmers to develop image processing applications for execution on homogeneous distributed memory MIMD-style multicomputers. As a result, this thesis addresses the following fundamental research issue: how to design a sustainable, yet efficient, software architecture for parallel image processing, that provides the user with a *fully sequential* programming model, and hides all parallelization and optimization issues from the user completely.

1.3 Thesis Outline

In the past, several parallelization tools have been described that, to a certain extent, serve as an aid to non-expert parallel programmers. As discussed extensively in Chapter 2, such tools generally suffer from fundamental problems that make them unsuitable as an acceptable long-term solution for the image processing community. Most importantly, the tools often are provided with a programming model that does not match the image processing researcher's frame of reference. In addition, efficiency of parallel execution is often far from optimal. Also, it is often hard to incorporate extensions to deal with new hardware developments and additional user requirements. To overcome these problems, Chapter 2 proposes a new and innovative software architecture for *user transparent parallel image processing*, that excludes its users from having to learn *any* skills related to parallelization and performance optimization.

In Chapter 3, we give a detailed account of the software architecture's design philosophy. We focus on implementing the architecture such that code redundancy is avoided as much as possible, and efficiency of execution is guaranteed. We demonstrate that the presented design philosophy allows for long-term architecture sustainability, as well as close-to-optimal performance.

In Chapter 4, we indicate how to apply a simple analytical performance model in the process of automatic parallelization and optimization of complete image processing applications. To this end, we present a high level abstract parallel image processing machine (*APIPM*), designed to capture typical run-time behavior of parallel low level image operations. From its instruction set, a high level performance model is obtained that is applicable to a relevant class of parallel platforms.

Chapter 5 addresses the problem of accurate cost estimation of the communication primitives applied in our software architecture. It is observed that existing communication models are not powerful enough to serve as a basis for automatic and optimal domain decomposition of the image data structures applied in typical applications. To overcome this problem, the specific capabilities of the applied communication primitives are combined into a new, more powerful performance model (*P-3PC*).

Chapter 6 deals with the problem of the automatic conversion of any sequential image processing application into a correct and efficient parallel version. To this end, we define a simple finite state machine specification that guarantees the conversion process to be performed correctly at all times. As the issue of automatic optimization of complete applications is the central problem our software architecture for user transparent parallel image processing is confronted with, Chapter 6 combines all of the results obtained in Chapters 3, 4, and 5.

In Chapter 7, we give an assessment of the software architecture's effectiveness in providing significant performance gains. We describe the implementation and automatic parallelization of three well-known example applications that contain many operations commonly applied in image processing. In addition, we investigate how well the performance obtained with our software architecture compares to that of reasonable hand-coded implementations.

Finally, in Chapter 8 we summarize the results of this research, and present our view on the developed architecture for user transparent parallel image processing.

Chapter 2

A Sequential Programming Model for Efficient Parallel Image Processing*

"O Freunde, nicht diese Töne!
Sondern laßt uns angenehmere anstimmen..."

Ludwig van Beethoven - *Symphony No. 9 "Choral"* (1824)

Parallel and distributed computing architectures, whose performance far exceeds that of traditional sequential systems, have been available for decades. As an example, the development of the Illiac IV [12], a machine commonly seen as the first true parallel system, started as early as 1965. In recent years, high performance computing systems have become more and more widespread, especially with the advent of highly flexible Field-Programmable Gate Arrays (FPGAs [18, 24, 66]) and relatively cheap Beowulf clusters [7, 157]. Also, specialized digital signal processing (DSP) devices and dedicated hardware architectures have become widely available [48, 91, 127].

As discussed in Chapter 1, the processing power as provided by parallel and distributed systems is essential for many image processing applications. Also, it has been recognized for years that the application of parallelism in imaging can be highly beneficial [161]. As a result, collaboration between the research communities of high performance computing and imaging has been commonplace, and typically resulted in specialized hardware configurations (e.g., see [47, 65, 88, 89, 107]) capable of efficiently executing domain-specific routines [19, 32, 134]. Yet, in spite of the importance of these achievements, the application of parallelism in imaging research is not widespread.

*This chapter contains portions of our paper as appeared in *Proceedings of the 15th International Conference on Pattern Recognition (ICPR 2000)* [140]. An extended version of this chapter is to appear in *Concurrency and Computation: Practice and Experience* [146].

Primarily, we ascribe the rather small user base of parallel computing within the image processing research community to the high threshold associated with the use of high performance computing architectures. One determinative factor for the existence of the threshold is the relatively high cost involved in using a specific parallel machine. In general, the image processing community can not afford to acquire and maintain such systems, and has to rely on hardware and support provided by the computing community. More importantly, the threshold exists due to a common characteristic of parallel and distributed systems, namely: they are much harder to program than sequential computers. Although several attempts have been made to alleviate the problem of software design for parallel and distributed systems, as of yet no solution is available that has found widespread acceptance.

As will be discussed extensively in this chapter, the latter problem is due to the fact that no efficient parallelization tool exists that is provided with a programming model that matches the entrance level of the average image processing practitioner. Most existing software development tools require the user to explicitly identify the available parallelism, often at a level of detail beyond the expertise and interest of most image processing researchers. Hence, it is essential to provide an alternative tool that offers a more 'familiar' programming model.

In this chapter we argue that a parallelization tool for the image processing research community is acceptable only if it hides all parallelism from the application programmer, and produces highly efficient code in most situations. Stated differently, we argue that a programming model is considered 'familiar' only, if it offers complete *user transparent parallel image processing*.

Several solutions have been described in the literature that allow for user transparent implementation of high performance image processing applications. In all cases, solutions are being provided in the form of an extensive software library containing parallel versions of fundamental image processing operations. The solutions, however, all suffer from one of several obstacles for widespread acceptance. Most significantly, the efficiency of parallel execution of complete applications often is far from optimal. In addition, the provided software library often does not incorporate a sufficiently high level of *sustainability*, thus dramatically reducing the chance on long term success.

Given these observations, the primary research issue addressed in this chapter is: How to provide the average image processing practitioner with a fully sequential programming model that allows for implementation of efficient parallel imaging applications such that the user is shielded from all issues related to parallelization and performance optimization? The second research issue addressed here is the following: How to incorporate such sequential programming model in an efficient parallelization tool that allows its developers to respond to changing demands quickly and elegantly?

In this chapter we propose a complete software architecture for user transparent parallel image processing that is specifically designed to deal with these issues. We discuss the requirements put forward for such programming tool, and provide a general overview of the architecture's constituent components. Essentially, this overview serves as a roadmap for the remaining chapters of the thesis.

This chapter is organized as follows. Section 2.1 presents a list of requirements a potential target hardware architecture should adhere to for it to be used in image

processing research. Based on the requirements one particular class of platforms is indicated as being most appropriate. In Section 2.2 the notion of *user transparency* is introduced, and used as a basis for an investigation of available tools for implementing parallel imaging applications on the selected set of target platforms. As the investigation shows that no existing development tool is truly satisfactory, our new software architecture for user transparent parallel image processing is introduced in Section 2.3. Concluding remarks are given in Section 2.4.

2.1 High Performance Computing Architectures

A parallelization tool intended as a programming aid in imaging research is more likely to find widespread acceptance if it is targeted towards a machine that is favored by the image processing research community. Implicitly (i.e., by ignoring inappropriate architectures), the imaging community has defined several requirements for such machines. These are formulated as follows:

- *Wide availability.* To ensure that the imaging community at large can benefit from a parallelization tool, it is essential that the target platform is widely available. Less popular or experimental architectures tend to suffer from a lack of continuity, thus hindering the ever present desire for hardware upgrades.

- *Ease of accessibility.* The target platform should be easily accessible to the image processing practitioner. This refers to the manner in which one logs on to the machine, how programs are to be compiled and run, and to the ease by which a set of processing elements is obtained. The last issue is particularly important where multiple users share a pool of processing elements.

- *Unrestricted programmability.* The hardware platform should not restrict the application programmer. It should be capable of executing the various operations commonly used as a basis in image and video processing applications.

- *Ready upgradability.* It is essential that the software developed for the target platform should be executable after each upgrade to the next generation of the same architecture. In other words, the desired continuity of the target platform requires a high degree of backward compatibility.

- *High efficiency.* The target platform should be capable of obtaining significant performance gains, especially for the most common imaging operations. If no significant improvements are to be expected, the process of accessing a parallel machine, and implementing and optimizing code for it, would be useless.

- *Low cost.* Even when significant speedups are to be expected, the financial burden of executing imaging software on the target platform should be kept to a minimum. As high performance computing is not a goal in itself in imaging research, the amount of money that may be spent on computing resources is small compared to the amount of money that flows to more fundamental research.

We are aware of the fact that additional requirements may hold in other application areas. Here, we deem such requirements to be either inherent to parallel systems in general (such as the desire for hardware scalability), or unimportant to most image processing practitioners (such as the amount of control over the structure, processing elements, operation, and evolution of a particular parallel system). Also, for specific image processing research directions additional requirements may be of significant importance. For example, in certain application areas strict limitations may be imposed on the target platform's size, or the amount of power consumption. In this thesis, however, we restrict ourselves to the list as presented here, as this represents the set of general requirements that holds for most image processing research areas.

Favoring Beowulf-type Commodity Clusters

As described in [33, 78, 146], several machines in the classes of *general purpose* MISD-, SIMD-, and MIMD-style parallel architectures (Flynn [49]) are potential candidates for high speed execution of image processing applications. Also, many *special purpose* architectures (e.g., ASICs [4, 98], FPGAs [24, 46], DSPs [47, 48]), as well as several enhanced general purpose CPUs [42, 116, 121], have been designed to obtain even higher performance for specific image processing tasks [33].

Irrespective of the significance of these systems, one architecture type stands out as particularly interesting for our purposes, i.e. the class of *Beowulf-type commodity clusters* [7, 157]. As one of the original designers of this type of architectures, Thomas Sterling, describes in a guest editorial on the clusters@top500 website [30], Beowulf-type systems are particularly important because "it is quite possible that by the middle of this decade clusters in their myriad of forms will be the dominant high-end computing architecture." Indeed, a strong trend in high performance computing is the growing use of commodity clusters, and many such systems are currently installed at research institutes and in commercial environments around the world.

Apart from being widely available, clusters often are made easily accessible to researchers from outside the computing community. Expected cooperation between multiple research disciplines often is the determinative factor in obtaining funding for such computer systems in the first place. In addition, the general-purpose nature of the constituent computing nodes fully adheres to the requirement of 'unrestricted programmability'. In fact, the bulk of all image processing research is currently being performed on similar computing nodes traditionally employed in a stand-alone manner. Also, a major advantage of the use of personal computers as constituent components is a long term continuity combined with 'ready upgradability'.

The single characteristic that makes a cluster favorable over other systems, however, is the emphasis on price-performance. As Sterling states in the same editorial: "for many application types, commodity clusters will deliver better, by even orders of magnitude in many cases, price-performance with respect to alternative systems". From these properties, in combination with the fact that many references exist that show significant performance gains for a multitude of different image processing applications (e.g., see [75, 79, 93, 159]), we conclude that clusters constitute the most appropriate target platforms for our specific needs.

2.2 Software Development Tools

Apart from its design and capabilities, the (commercial) success of any computer architecture significantly depends on the availability of tools simplifying software development. As an example, for many users it is often desirable to be able to develop programs in a high-level language such as C or C++. Unfortunately, and in contrast with general-purpose sequential systems, for many of the hardware architectures referred to in Section 2.1 available high-level language compilers often have great difficulties in generating assembly code that makes use of the machine's parallel capabilities effectively. As a result, for highest performance the programmer often must optimize the critical sections of a program by hand.

Whereas assembly coding or hand-optimization may be reasonable for a small group of experts, most users prefer to dedicate their time to describing *what* a computer should do rather than *how* it should do it. Consequently, many programming tools have been developed to alleviate the problem of low level software design for parallel and distributed systems. In all cases such tools are provided with a programming model that abstracts from the idiosyncrasies of the underlying parallel hardware. The small user base of parallel computing in the imaging community indicates, however, that no existing parallelization tool incorporates a level of abstraction that truly matches the image processing researcher's frame of reference.

The ideal solution would be to have a parallelization tool that abstracts from the underlying hardware completely, allowing users to develop *optimally efficient* parallel programs in a manner that requires *no additional effort* in comparison to writing purely sequential software. Unfortunately, no such parallelization tool currently exists and due to the many intrinsic difficulties it is commonly believed that no such tool will be developed ever at all [17]. However, if the ideal of 'obtaining optimal efficiency without effort' is relaxed somewhat, it may still be possible to develop a parallelization tool that constitutes an acceptable solution for the image processing research community. The success of such a tool largely depends on the amount of *effort* requested from the application programmer and the level of *efficiency* obtained in return.

The graph of Figure 2.1 depicts a general classification of parallelization tools based on the two dimensions of effort and efficiency. Here, the efficiency of a parallelization tool is loosely defined as the average ratio between the performance of any image processing application implemented using that particular tool and the performance of an optimal hand-coded version of the same application. Similarly, the required effort refers to (1) the amount of *initial learning* needed to start using a given parallelization tool, (2) the additional expense that goes into obtaining a parallel program that is *correct*, and (3) the amount of work required for obtaining a parallel program that is particularly *efficient*. In the graph, the maximum amount of effort the average image processing practitioner generally is willing to invest into the implementation of efficient parallel applications is represented by `THRESHOLD 1`. The minimum level of efficiency a user generally expects as a return on investment is depicted by `THRESHOLD 2`. To indicate that the two thresholds are not defined strictly, and may differ between groups of researchers, both are represented by somewhat fuzzy bars in the graph of Figure 2.1.

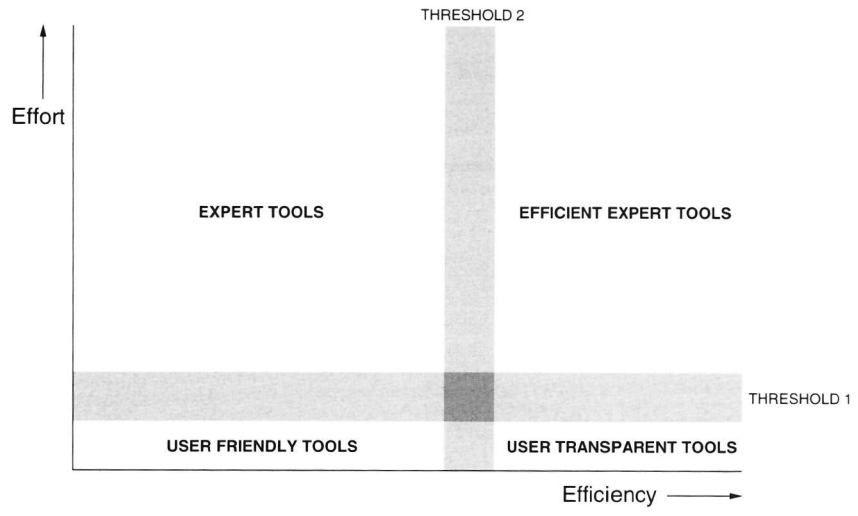


Figure 2.1: *Parallelization tools: effort versus efficiency. User transparent tools are considered both user friendly and sufficiently efficient.*

In this thesis, each tool that is considered both 'user friendly' and 'sufficiently efficient' is referred to as a tool that offers full *user transparent parallel image processing*. Apart from adhering to certain levels of requested effort and obtained efficiency, an important additional feature of any user transparent tool is that it does not require the user to fine-tune any application in order to obtain particularly efficient parallel code (although the tool may still allow the user to do so). Based on the above considerations, we conclude that *a parallelization tool constitutes an acceptable solution for the image processing community only, if it can be considered fully user transparent.*

One may argue that the thresholds in Figure 2.1 are not straight lines in each of the two dimensions, but are better combined in a single diagonal (or curved) line. This would be reasonable, as for a small amount of obtained efficiency the user is probably not prepared to invest as much effort as for a much higher level of efficiency. The presented classification is still valid, however, as we argue that it should not be *required* from the user to invest any additional effort to obtain higher efficiency.

2.2.1 General Purpose Parallelization Tools

The following gives an overview of the most significant development tools that (a.o.) can be used for implementing image processing applications on clusters. For each tool we discuss the level of abstraction incorporated in the programming model, and assess to what extent it adheres to the properties of full user transparency. The discussion starts with an overview of general-purpose parallelization tools, and is followed by an overview of tools designed specifically for developing high performance image processing applications.

Message Passing Libraries

Good examples of tools from the set of *efficient* programming aids for *experts* in parallel computing are the many software libraries providing *message passing* functionality [6]. Message passing is a programming paradigm based on the concept of processes that explicitly communicate data. It is mainly intended for programming distributed memory MIMD-style multicomputers, but the paradigm applies to shared memory machines as well. Many efficient and portable message passing systems have been described in the literature [102], but the sets of library routines provided by PVM (Parallel Virtual Machine [53]) and MPI (Message Passing Interface [61, 104, 105]) have become the most widely used [54, 67].

Parallel programming on the basis of message passing requires the programmer to personally manage the distribution and exchange of data, and to explicitly specify the parallel execution of code on different processors. Although this approach often produces highly efficient parallel programs, even for expert programmers it is difficult to do correctly [29]. This is due to the fact that message passing tools do not provide explicit support for the design and implementation of parallel data structures. Also, deadlocks are introduced easily, and debugging is hard under critical dependencies in the relative timing of events. Due to these problems, message passing is often referred to as the "assembly language of parallel computing", since it offers "a means for expressing parallel computation in an often painstaking, low-level, error-prone manner" [23]. Given these observations, we conclude that message passing is not the programming paradigm of choice for the average image processing researcher.

Shared Memory Specifications

As message passing was intended for client/server applications running across a network, PVM and MPI include costly semantics (e.g., the assumption of wholly separate memories) that are often not required on parallel systems with a globally addressable memory. To provide a simpler, yet efficient, and portable approach to implementing parallel programs, several shared memory specifications have been proposed, such as CRL [76] and Midway [15]. OpenMP [25, 119], which consists of a set of compiler directives, library routines, and environment variables to specify shared memory parallelism in Fortran and C/C++ programs, is the most commonly used.

Although a cluster does not fit in the class of shared-memory architectures, it is still relevant to include shared memory specifications in this evaluation. This is because shared memory specifications can be implemented on top of MPI, albeit at the cost of higher latencies [41]. Also, the provided programming paradigm is generally believed to be much simpler than MPI [57, 113],

One of the major advantages of shared memory specifications is that it is easy to incrementally parallelize sequential code. For non-expert programmers, however, it is still difficult to write efficient and scalable programs. In addition, the presence of both shared and private variables often causes confusion. As a result, the amount of effort requested from the average user still exceeds `THRESHOLD 1` in Figure 2.1. Therefore we conclude that shared memory specifications fall in the set of 'efficient expert tools' as well, and do not adhere to the requirements of full user transparency.

Extended High-Level Languages

An alternative to the library approach as followed by MPI and OpenMP is to provide a small set of modifications and/or extensions to an existing high-level programming language. Probably the most popular example of a language that has adopted this approach is HPF (High Performance Fortran [97]). A similar approach is followed in SPAR [128, 129], which is one of the many extended, parallel versions of Java. Also, many alternative extensions and modifications to C++ exist [171], of which Compositional C++ [26] and Mentat [60] are the most significant examples.

Irrespective of language design and compilation issues, for users of such languages the most important problem is that it is often required to understand in what situations the compiler can produce efficient executable code. For example, HPF requires that the distribution of data is specified separately from the routines operating on that data. Consequently, a mismatch between data distribution and functionality is easily introduced, possibly resulting in reduced performance due to huge amounts of unnecessary communication. As state-of-the-art compilers are not capable of detecting all such non-optimal behavior automatically [8, 17], much of the efficiency of parallel execution is still in the hands of the application programmer. As a result, the amount of effort a non-expert user must invest into writing efficient parallel codes in an extended high-level language also exceeds `THRESHOLD 1` in Figure 2.1.

Parallel Languages

Rather than extending an existing sequential language, it is also possible to design an entirely new parallel programming language from scratch. Considering parallelism directly in the design phase of a concurrent language offers a better chance of obtaining a clean and unified parallel programming model. Also, this approach facilitates implementation of efficient compiler optimizations, and the development of effective debugging tools. For these reasons, many parallel languages have been described in the literature (e.g., Ada [13], Occam [77], Orca [8, 137, 138], and Parlog [58]).

Despite years of intensive research, no parallel language has truly found widespread acceptance, either in the imaging community or elsewhere. One reason is that it appears to be difficult to design language features that are both generally applicable and easy to use [120]. A more important reason is that most scientific programmers are reluctant to learn an entirely new program development philosophy, or unfamiliar language constructs. As the parallelism in a parallel language is always explicit, and fine-tuning is often an inherent part of the program development process, we conclude that the amount of effort required from the average user generally is too high.

Fully Automatic Parallelizing Compilers and Parallelizing Pre-Compilers

As opposed to the parallelization tools discussed so far, an efficient *automatic parallelizing compiler* would constitute an ideal solution. It would allow programmers to develop parallel software by using a sequential high-level language *without* having to learn additional parallel constructs or compiler directives [10]. However, a fundamental problem is that many user-defined algorithms contain data dependencies that

prevent efficient parallelization. This problem is particularly severe for languages supporting pointers [2]. In addition, techniques for automatic dependency analysis and algorithm transformation are still in their infancy. Although interesting solutions have been reported that require the user to be conservative in application development (e.g., to allow efficient parallelization of loop constructs [52]), fully automatic parallelizing compilers that can produce efficient parallel code for any type of application do not exist — and a real breakthrough is not expected in the near future [17].

As an alternative, effort is currently being put into semi-automatic tools (such as FORGE [5]) that require the programmer to help the compiler interactively in the parallelization process. Although, in principle, this approach could allow user transparent implementation of parallel imaging applications, it can not be considered an acceptable solution. This is because the approach does not eliminate the burden of specifying the available parallelism; it merely pushes the problem forward to a later stage in the program development process.

2.2.2 Tools for Parallel Image Processing

The regular evaluation patterns in many low level image processing operations often make it easy to determine how to parallelize such routines efficiently. Also, because many different image operations incorporate similar data access patterns, a small number of alternative parallelization strategies often need to be considered. These observations have led to the creation of software development tools that are specifically tailored to image processing applications. Such tools may provide higher abstraction levels to the user than general-purpose tools, and are potentially much more efficient as important domain-specific assumptions often can be incorporated.

Programming Languages for Parallel Image Processing

One approach to integrating domain-specific knowledge is to design a programming language for parallel image processing specifically. Apply [64, 164] was one of the first attempts in this direction. It is a simple, architecture-independent language restricted to *local* image operations, such as edge detection, smoothing, and point operations. It is based on the observation that many operations follow a stereotypical form:

```
for each row
   for each column
      produce an output pixel based on a window of pixels around
      the current row and column in the input image
```

Apply exploits this idea by requiring the programmer to write only the innermost 'per pixel' portion of the computation. The iteration is then implicit and can easily be made parallel. Apply's restricted programming model allows easy implementation of quite an extensive set of operations. The programmer simply has to describe the program in terms of the smallest meaningful unit — namely, a window taken around a pixel in an image. Because a program is specified in this way, the compiler needs

only to divide the images among processors and then iterate the Apply program over the image sections allocated to each processor. Despite the fact that the language was capable of providing significant speedups for many applications, the programming model proved to be too restricted for practical use.

In a different language, called Adapt [165], the basic principles of Apply are extended to incorporate *global* operations as well. In such operations an output pixel can depend on many or all pixels in the input image. Adapt is based on the split-and-merge programming model, in which data structures are split according to data position, and separately computed adjacent results are then merged. The programmer has to describe both the operation to be performed at every pixel of the image (as in Apply), as well as a combining operation to merge two results produced independently at different processors. Although the language certainly allows for an efficient parallel implementation of many important image processing applications, the programming model is not ideal. This is because the programmer is personally responsible for data partitioning and merging, albeit at quite a high level. For this reason we categorize Apply as an 'efficient expert parallelization tool' as well. Yet, it may constitute an acceptable solution for quite a large group of users.

An alternative approach is taken in a language called IAL (Image Algebra Language [35, 37]). IAL is based on the abstractions of Image Algebra [131], a mathematical notation for specifying image processing algorithms. IAL provides operations at the complete image level, with no access to individual pixels. For example, the Sobel edge detector is implemented in IAL as a single statement:

```
OutputIm := (abs(InputIm ⊕ S_h) + abs(InputIm ⊕ S_v)) >= threshold;
```

where S_h and S_v are the horizontal and vertical Sobel masks, and $\oplus$ represents convolution. The language proved to be useful for a wide range of tasks, but was limited in its expressive power. Two extended versions of IAL, I-BOL [20] and Tulip [155] provide a more flexible and more powerful notation. The languages permit access to data at either the pixel level or at the neighborhood level, without being architecture-specific. Although the languages hide all parallelism from the user, a major disadvantage is that it proved to be difficult to incorporate a global application optimization scheme to ensure efficiency of complete programs at all times. Another disadvantage is that the syntax of the languages differs quite somewhat from C and C++ — arguably the most popular languages applied in the image processing community.

Parallel Image Processing Libraries

An alternative to the language approach is to provide an extensive set of parallel image processing operations in library form — possibly as part of a complete framework that deals with additional issues, such as global application optimization. In principle, this approach allows the programmer to write applications in a familiar sequential language, and make use of the abstractions as provided by the library. Due to the relative ease of implementation, many parallel image processing libraries have been described in the literature, and here we will shortly discuss the most important ones.

One particularly interesting data parallel library implementation is described by Taniguchi et al. [159]. This software platform is applicable to both SIMD- and MIMD-style architectures, and incorporates a data structure abstraction known as DID, for *distributed image data*. The DID-abstraction is intended as an image data type declaration, without exposing the actual distribution of data. For example, a DID structure for binary image data may be declared as:

```
Image2D_Binary bimage(Horizontal, "pict1.jpg");
```

to indicate that a binary image `"pict1.jpg"` is read into a horizontally distributed image data structure, which can be referred to through `bimage`. Although a DID declaration is easy to understand for programmers unfamiliar to parallel computing, it has the disadvantage of making the user responsible for the type of data distribution.

Another library-based approach applicable to both SIMD- and MIMD-style architectures is developed by Olk et al. [118]. The library provides a fully sequential interface to the user, and incorporates data parallel data structure abstractions such as images, kernels, neighborhoods, queues, buckets, etcetera. The programmer addresses a data structure as a single entity, with no concern of the implementation and parallel execution of an operation. However, to obtain efficient executables the user needs to implement in Compositional C++ [26] (see Section 2.2.1). Clearly, this is approach is not ideal, as it still requires the programmer to personally identify part of the available parallelism.

The library-based environment described by Jamieson et al. [73, 74, 75, 168] also provides a fully sequential interface to the user. At the heart of the environment is a set of algorithm libraries, along with abstract information about the performance characteristics of each library routine. In addition, the environment contains a dynamic scheduler for optimization of full applications, an interactive environment for developing parallel algorithms, and a graph matcher for mapping algorithms onto parallel hardware. Although this environment proved to be quite successful, its sustainability proved to be problematic. Partially, this is because it is required to provide *multiple* implementations for an algorithm, one for each target parallel machine.

One data parallel environment that indeed can be considered fully user transparent is developed by Lee et al. [93]. An interesting aspect of this work is that it incorporates simple performance models to ensure efficiency of execution of complete applications. However, the environment is too limited in functionality to constitute a true solution, as it supports point operations and a small set of window operations only. Two similar environments, presented in [79, 80, 81] and [86, 87] respectively, are much more extensive in functionality. However, in both cases the performance models as designed in relation with the library operations are not used as a basis for optimization of complete programs, but serve as an indication to library users only.

An interesting environment based on the abstractions of Image Algebra [131], that to a large extent adheres to the requirements of user transparency, is described in [109]. It is targeted towards homogeneous MIMD-style multicomputers, and is implemented in a combination of C++ and MPI. One of the important features of this environment is the so-called *self-optimizing class library*, which is extended automatically with optimized parallel operations. During program execution, a syntax

graph is constructed for each statement in the program, and evaluated only when an assignment operator is met. At first execution of a program, each syntax graph is traversed, and an instruction stream is generated and executed. In addition, any syntax graph for combinations of primitive instructions (i.e., those incorporated as a single routine within the library) is written out for later consideration by an off-line optimizer. On subsequent runs of the program a check is made to decide if an optimized routine is available for a given sequence of library calls. An important drawback of this approach, however, is that it may guarantee optimal performance of sequences of library routines, but not necessarily of complete programs.

The MIRTIS system, described in [108], is the most efficient and extensive library-based environment for user transparent parallel image processing designed to date. MIRTIS is targeted towards homogeneous MIMD-style architectures, and provides operations at the complete image level. Programs are parallelized automatically by partitioning sequential data flows into computational blocks, to be decomposed in either a spatial or a temporal manner. Issues related to data decomposition, communication routing, and scheduling are dealt with by using simple performance models. In the modeling of the execution time of a certain application, MIRTIS relies on empirically gathered benchmarks. Although, from a programmer's perspective, MIRTIS constitutes an ideal solution, its implementation suffers from poor maintainability and extensibility. Also, the provided MIRTIS implementation suffers from reduced portability as the applied communication kernels are too architecture specific.

From this overview we conclude that, although several library-based user transparent systems exist, none of these is truly satisfactory. As indicated in the discussion, this is because it is not sufficient to offer user transparency *as is*. Issues relating to the *design* and *implementation* of a parallelization tool, such as maintainability, extensibility, and portability of the provided software library, play an important role as well. A discussion of these issues follows in the remainder of this chapter.

2.2.3 Discussion

In Figure 2.2 we have positioned all classes of parallelization tools presented in this section in a single effort-efficiency graph similar to that of Figure 2.1. The figure shows that the amount of effort required for using any type of *general-purpose* parallelization tool generally exceeds `THRESHOLD 1` (the class of automatic parallelizing compilers being the only exception). Also, the higher the efficiency provided by such general-purpose tool, the higher the amount of effort required from the application programmer. Although the introduction of domain-specific knowledge reduces the required amount of user effort, parallel image processing languages are generally still too specialized for widespread acceptance. From the two classes of tools that are considered 'user-friendly' by the image processing community (i.e., automatic parallelizing compilers and parallel image processing libraries), only a small subset of all library-based tools provides a sufficiently high level of efficiency as well.

Despite the fact that some of the library-based systems adhere to all requirements of user transparency (especially those described by Lee et al. [93], Moore et al. [108], and Morrow et al. [109]), none of these has found widespread acceptance. One may

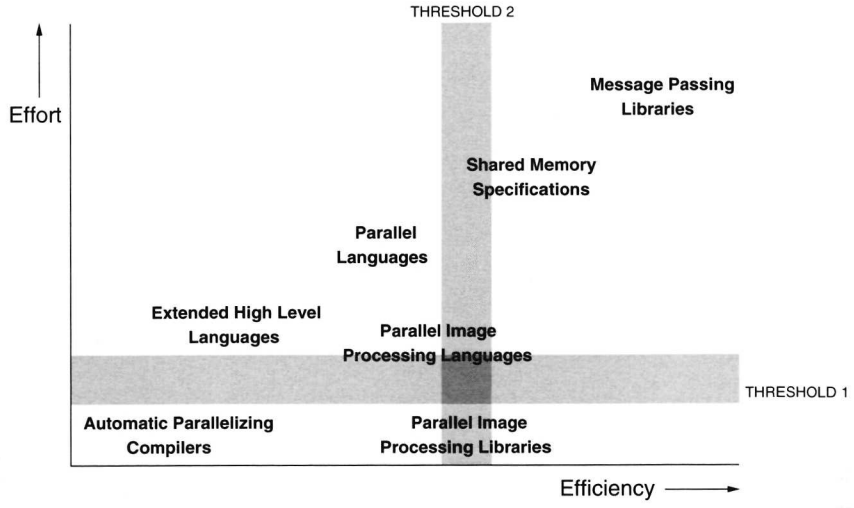


Figure 2.2: *Generalized view of effort versus efficiency of existing parallelization tools. From the set of tools only a subset of all parallel image processing libraries can be considered truly user transparent.*

argue that this is due to the fact that they are still relatively new, and may need some more time to make a significant impact on the imaging community. We feel, however, that the tools still do not constitute a solution that is acceptable *on the long term.*

As we have discussed extensively in the previous sections, user transparency in itself is the decisive property that matches a tool's programming model to the image processing researcher's frame of reference. In this respect, any tool that adheres to the requirements of user transparency is acceptable in that it can always be used immediately, without much effort from the application programmer. However, a parallelization tool is not a static product. It is essential for such tool to be able to deal with new hardware developments and additional user requirements. If the design of a parallelization tool makes it ever more difficult or even impossible for its developers to respond to changing demands quickly and elegantly, users will loose interest in the product almost immediately.

If we refer back to the graph of Figure 2.1, perpendicular to the two dimensions of effort and efficiency we can add a third axis that represents a tool's level of *sustainability*. This term incorporates all issues relating to the extensibility, maintainability, applicability, and portability of a given parallelization tool, and indicates how easily a tool can be adapted to changing demands and environments. As before, a critical threshold can be identified for the level of sustainability, below which no tool is expected to survive on the long term. We feel that none of the existing user transparent tools incorporates an acceptable sustainability level as well. For this reason we have designed a new parallelization tool that, apart from adhering to the requirements of full user transparency, also offers a sufficiently high level of sustainability.

2.3 A Sustainable Software Architecture for User Transparent Parallel Image Processing

The discussion of the applicability of existing hardware and software architectures in the field of image processing research has led to several important conclusions. First, the most appropriate class of hardware architectures to be applied in image processing research is that of Beowulf clusters — most importantly due to its emphasis on price-performance. Second, software development tools based on a library of pre-parallelized routines offer a solution that is most likely to be acceptable to the image processing community — especially because it has shown to be possible to provide such tool with a programming model that offers full user transparency. Finally, no user transparent tool currently exists that indeed provides an acceptable long term solution, as none incorporates a sufficiently high level of sustainability.

In this section we present an overview of our new library-based architecture for user transparent parallel image processing on homogeneous clusters. Due to its innovative design we expect the architecture to constitute an acceptable solution for the image processing community on the long term.

2.3.1 Architecture Requirements

We argue that a library-based software architecture, which is to serve as a parallelization aid for the image processing research community, must adhere to the following list of requirements:

I. *User transparency.* As discussed in Section 2.2, user transparency refers to a combination of 'user friendliness' and 'high efficiency'. For a library-based parallelization tool, this terminology translates to the following two requirements:

   1. *Availability of an extensive sequential API.* To ensure that the parallel library is of great value to the image processing community, it must contain an extensive set of data types and associated operations commonly applied in image processing research. The application programming interface (API) should disclose as little as possible information about the library's parallel processing capabilities. Preferably, the API is made identical to that of an existing *sequential* image processing library.

   2. *Combined intra-operation efficiency and inter-operation efficiency.* It is essential for the software architecture to provide significant performance gains for a wide range of image processing application types. For this reason it is required to obtain a level of efficiency that generally compares well to that of 'reasonable' hand-coded parallel implementations. Efficiency, in this respect, refers to the execution of each library operation in isolation (*intra-operation efficiency*), as well as to the execution of multiple operations applied in sequence (*inter-operation efficiency*).

II. *Long term sustainability.* To ensure longevity, the design and implementation of the software architecture must be such that extensions are easily dealt with. In this respect, long term sustainability refers to the following four requirements:

3. *Architecture maintainability.* To minimize the coding effort in case of changing demands and environments, care must be taken in the architecture's design to avoid unnecessary code redundancy, to enhance operation reusability. In this respect, it is preferable to implement any set of operations with similar behavior as a single generic routine, to be instantiated at will to obtain the desired functionality. Also, to avoid implementing operations for all data types generic implementations are preferred.
4. *Architecture extensibility.* As no library can contain all functionality applied in image processing research, it is required to allow the user to insert new operations. In case an additional operation maps onto a generic operation present in the library, insertion should be straightforward, not requiring any parallelization effort from the user.
5. *Applicability to homogeneous Beowulf clusters.* As we have identified clusters as the most appropriate type of hardware architecture for image processing research (see Section 2.1), the complete software architecture must be applicable to this type of machines. All general and distinctive properties of such machines can therefore explicitly be incorporated in the implementation of the software architecture. Optimized functionality for any other machine type should not be incorporated.
6. *Architecture portability.* To ensure portability to all target machines it is essential to implement the software architecture in a high-level language such as C or C++. For any constituent component in a cluster a high quality C or C++ compiler is generally available — and upgrades are released frequently. Although the properties of Beowulfs can be incorporated in all implementations, care should be taken not to incorporate any assumptions about a specific interconnection network topology.

2.3.2 Architecture Overview

The complete software architecture consists of six components (see Figure 2.3). This section presents a general overview of each of the components, and design choices are related to the requirements of Section 2.3.1.

Component 1: Parallel Image Processing Library

The core of our software architecture consists of an extensive software library of data types and associated operations commonly applied in image processing research. In accordance with the first requirement of Section 2.3.1, the library's application programming interface is made identical to that of an existing sequential image processing library: Horus [84]. More specifically, rather than implementing a completely new

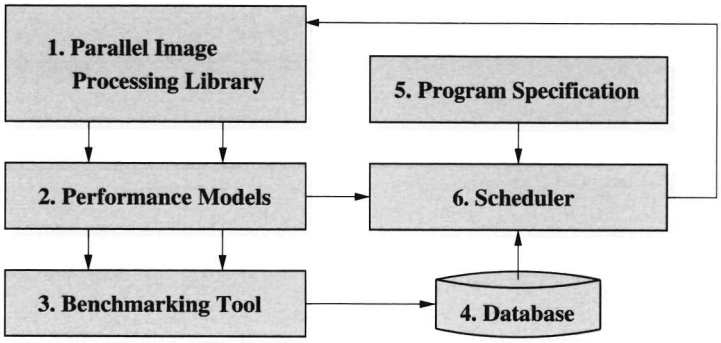


Figure 2.3: *Simplified architecture overview.*

library from scratch, the parallel functionality is integrated with the Horus implementation in such a manner that all existing sequential code remains intact. Apart from reducing the required parallel implementation effort, this approach has the advantage that the important properties of the Horus library (i.e., maintainability, extensibility, and portability) to a large extent transfer to the parallel version of the library as well.

Similar to other libraries discussed in Section 2.2.2, the sequential Horus implementation is based on abstractions of Image Algebra [131], a mathematical notation for specifying image processing algorithms. Image Algebra is an important basis for the design of an extensive, maintainable, and extensible image processing library, as it recognizes that a small set of *operation classes* can be identified that covers the bulk of all commonly applied image processing functionality. Within the Horus library each such operation class is implemented as a *generic algorithm*, using the C++ *function template mechanism* [158]. Each operation that maps onto the functionality as provided by such algorithm is implemented by instantiating the generic algorithm with the proper parameters, including the function to be applied to the individual data elements. From this, it follows that the desired architectural properties of maintainability, extensibility, and portability, constitute an integral aspect the Horus design. As will be discussed in more detail in Chapter 3, the Horus library also covers a large majority of all common image processing operations. As a result, Horus fully adheres to requirements 1, 3, 4, and 6 of Section 2.3.1.

In extending the Horus library for parallel operation we have focused on adhering to the remaining requirements 2 and 5: i.e., the architecture's efficiency and its applicability to Beowulfs. To this end, and also to have full control over the communication behavior of the library operations, the parallel extensions are implemented using MPI [104]. Also, to sustain a high maintainability level, each parallel image processing operation is implemented by concatenating data communication routines with sequential code blocks from the Horus library. In this manner, the source code for each sequential generic algorithm is fully reused in the implementation of its parallel counterpart, thus avoiding unnecessary code redundancy as much as possible. For a more detailed description of the library implementation, we refer to Chapter 3.

The design and implementation of the parallel library ensures that our parallelization tool adheres to all requirements of Section 2.3.1, with the exception of requirement 2. To also guarantee efficiency of execution of (1) operations that are applied in isolation, and (2) applications or algorithms that contain sequences of library operations, five additional architectural components are designed and implemented in close connection with the software library itself. These additional components are described in the remainder of this section.

Component 2: Performance Models

In contrast to other library-based environments (e.g., [75]), our library contains not more than one parallel implementation for each generic algorithm. To still guarantee intra-operation efficiency on all target platforms, the parallel generic algorithms are implemented such that they are capable of adapting to the performance characteristics of the parallel machine at hand. As an example, the manner in which data structures are decomposed at run time is not fixed in the implementations, as the efficiency of each decomposition type may differ for each specific target machine. Also, the optimal number of processing units may vary.

To make a machine's performance characteristics explicit, each library operation is annotated with a domain specific performance model. For applicability to clusters, the models are based on an abstract machine definition (the *APIPM*, or: Abstract Parallel Image Processing Machine) that captures the hardware and software aspects of image processing operations executing on such a system. An overview of the APIPM, as well as a formal definition of the APIPM-based models for *sequential* operation, is presented in Chapter 4. A detailed description of the model that captures the additional *communication* aspects of parallel execution is given in Chapter 5.

Component 3: Benchmarking Tool

Performance values for the model parameters are obtained by running a set of *benchmarking* operations that is contained in a separate architectural component. The combination of the high-level APIPM-based performance models and the specialized set of benchmarking routines allows us to follow a *semi-empirical modeling* approach, that has proven to be highly successful in other research as well (e.g., see [108, 172]). In this approach, essential but implicit cost factors are incorporated by performing actual experiments on a small set of sample data. Apart from its relative simplicity, the main advantage of the semi-empirical modeling approach is that it fully complies with the requirements of applicability and portability to clusters. The performance models and benchmarking results allow intra-operation optimization to be performed automatically, fully transparent to the user. This optimization is performed by the architecture's scheduling component, described below.

Chapter 4 gives a thorough description of the approach of semi-empirical modeling, as well as an overview of the benchmarking strategy applied for the measurement of sequential operations. An overview of the measurement strategy relating to the communication aspects of parallel execution is given in Chapter 5.

Component 4: Database of Benchmarking Results

All benchmarking results are stored in a database of performance values. Although the design and implementation of such database is of significant importance (especially in case it must be accessed frequently at run time), this topic is too far outside the scope of this thesis for extensive discussion.

Component 5: Program Specification

Apart from incorporating an intra-operation optimization strategy, to obtain high efficiency it is essential to perform inter-operation optimization (or: optimization across library calls) as well. As it is often possible to combine the communication steps of multiple library operations applied in sequence, the cost of data transfer among the nodes in a parallel machine generally can be reduced considerably. Our architecture performs inter-operation optimization in case global information is available on the order in which library operations are applied in a given application. Essentially, this information is obtainable from the original program code. As implementation of a complete parser is not an essential part of this research, however, we currently assume that a complete algorithm specification is provided in addition to the program itself. Such specification closely resembles a concatenation of library calls, and does not require any parallelism to be introduced by the application programmer.

Component 6: Scheduler

Once the performance models, the benchmarking results, and the algorithm specification are available, a scheduling component is applied to find an optimal solution for the application at hand. The scheduler performs the tasks of intra-operation optimization and inter-operation optimization by removing all redundant communication steps, and by choosing: (1) the logical processor grid to map data structures onto (i.e., the actual domain decomposition), (2) the routing pattern for the distribution of data, (3) the number of processing units, and (4) the type of data distribution (e.g., broadcast instead of scatter).

As described in detail in Chapter 6, the scheduler's task of automatically converting any sequential image processing application into a correct and efficient parallel version, is performed on the basis of a simple finite state machine definition. First, the finite state machine allows for a straightforward and cheap run time method (called *lazy parallelization*) for communication cost minimization. If desired, the scheduler can be instructed to perform further optimization at compile-time. In this case, the finite state machine is used in the construction of an application state transition graph, that fully characterizes an application's run time behavior, and incorporates all possible parallelization and optimization decisions. As each decision is annotated with a run time cost estimation obtained from the APIPM-based performance models, the fastest version of the program is represented by the cheapest branch in the application state transition graph. In the library implementation of each parallel generic algorithm, requests for scheduling results are performed in order to correctly execute the optimizations prescribed by the application state transition graph.

2.4 Conclusions

In this chapter, we have investigated the applicability of existing hardware and software architectures in the field of image processing research. Based on a set of architecture requirements we have indicated that homogeneous Beowulf clusters constitute the most appropriate class of target platforms for application in image processing research. Apart from the fact that many references exist in the literature indicating significant performance gains for typical image processing applications executing on clusters, the foremost reason for favoring such architectures over other appropriate systems was found to be the fact that these deliver the best combination of price and performance.

Our investigation of *software tools* for implementing image processing applications on clusters has shown that library-based parallelization tools offer a solution that is most likely to be acceptable to the image processing research community. First, this is because such tools allow the programmer to write applications in a familiar programming language, and make use of the high level abstractions as provided by the library. More importantly, this is because library-based environments are most easily provided with a programming model that offers full *user transparency* — or, in other words: sufficiently high levels of 'user friendliness' and 'efficiency of execution'. Due to insufficient sustainability levels, no existing user transparent tool was found to provide an acceptable *long term* solution as well.

On the basis of these considerations we have proposed a new library-based software architecture for parallel image processing on clusters. We have presented a list of requirements such tool must adhere to for it to serve as an acceptable long term solution. In addition, we have given an overview of each of the architecture's constituent components, and we have touched upon the most prominent design issues for each of these. The architecture's innovative design and implementation ensures that it fully adheres to the requirements of user transparency and long term sustainability. Consequently, we believe our architecture for user transparent parallel image processing to constitute an acceptable long term solution for the image processing research community at large.

Chapter 3

Parallelizable Patterns in Low Level Image Processing Algorithms*

"One gets to the heart of the matter by a series of experiences in the same pattern, but in different colors."

Robert Graves (1895 - 1985)

As discussed in the previous chapter, a multitude of software libraries for parallel low level image processing has been described in the literature [75, 80, 93, 108, 109, 112, 118, 153, 159]. An important design goal in much of this research is to provide operations that have optimal efficiency on a range of parallel machines. In general, this is achieved by hard-coding a number of different parallel implementations for each operation, one for each platform. Unfortunately, the creation of a parallel library in this manner has several major drawbacks. First, manually creating multiple parallel versions of the many operations commonly applied in image processing research is a laborious task. Second, obeying to requests for library extensions becomes even more troublesome than in the sequential case. Third, as new target platforms are made available at regular intervals, code maintenance becomes hard — if not impossible — on the long term. Finally, with each library expansion it becomes ever more difficult to incorporate a single elegant optimization strategy that can guarantee intra-operation efficiency as well as inter-operation efficiency. For these reasons we take a different approach.

*This chapter combines our papers published in *Proceedings of the 15th International Parallel & Distributed Processing Symposium (IPDPS 2001)* [141] and *Parallel Computing* [149].

In the design of our parallel library we strive to minimize the implementation effort, without compromising on the efficiency of execution. The first step in achieving this goal is recognizing that there is a limited number of ways in which the pixels in an image can be processed to produce meaningful results. Important in this respect is the classification of low level image processing operations made in Image Algebra [131]. Originating from this classification, the sequential Horus image processing library [84] (which serves as a basis for the core component of our software architecture, see Section 2.3.2) provides a small set of so-called *algorithmic patterns*. As will be explained in this chapter, the primary importance of the algorithmic patterns is that each serves as a template operation for a large set of image processing operations with comparable behavior. Also, the algorithmic patterns abstract from the actual datatype each operation is applied upon, to avoid a combinatorial explosion of code that deals with all possible kinds of image datatypes.

The next important step in achieving our goal is recognizing that, for parallel implementation of each algorithmic pattern, much of the related sequential code can be reused. To that end, for each sequential algorithmic pattern present in the Horus library we have defined a so-called *parallelizable pattern*. Such pattern constitutes the maximum amount of code of an algorithmic pattern that can be performed both sequentially and in parallel — in the latter case without having to communicate to obtain data residing on other processing units.

The final step in reaching our goal is to implement all parallel operations such that they are capable of adapting to the performance characteristics of a parallel machine at hand. As machine-specific performance characteristics should not be incorporated explicitly in any library implementation, an additional automatic code optimization phase is required to be performed at compile time or even at run time.

Hence, apart from giving a detailed overview of the design philosophy of our software library, this chapter primarily focuses on the following research issue: How to implement a parallel image processing library such that code redundancy is avoided as much as possible, and efficiency of execution on all target platforms is guaranteed. We present a solution to the problem in the form of a *generic description* of parallelizable patterns. Based on the description, we show how parallel versions of many commonly used image processing operations are obtained by concatenating high-level communication routines, basic memory operations, and operations that constitute a specialization of a parallelizable pattern. We demonstrate that, apart from being relatively simple to implement, a parallel library built in this manner is extensible, easily maintainable, and still high in performance.

It is important to stress that this chapter does not touch upon the important topic of inter-operation optimization, that is, optimization across library calls. Parallel operations that are implemented on the basis of parallelizable patterns may still perform many unnecessary communication steps when applied as part of a complete image processing application. As a result, efficiency of execution may not be optimal. As indicated in the previous chapter, our complete software architecture deals with this problem by applying domain-specific performance models in combination with an additional, integrated scheduling tool. These issues are all outside the scope of this chapter, however, and are discussed in extensive detail in Chapters 4, 5, and 6.

This chapter is organized as follows. Section 3.1 gives an overview of the design philosophy of the sequential Horus library. Section 3.2 describes the manner in which parallelism is integrated in Horus. Section 3.3 discusses the programming paradigm adopted in all parallel implementations. Section 3.4 gives a generic description of parallelizable patterns, including a default parallelization strategy for image operations. To illustrate the use of parallelizable patterns, the implementation of two example operations is discussed in detail. Finally, conclusions are presented in Section 3.5.

3.1 Algorithmic Patterns: The Horus Approach

Whereas implementation of a single sequential image processing routine is often easy, creating a software library that is to contain an extensive set of such operations is notoriously hard. This is because image library users need operations that can be applied to a large number of (combinations of) different data structures, whose individual data elements in turn can be of many different types. More specifically: although two-dimensional image structures are most commonly used, the bulk of all library functionality also should be applicable to three- (or higher-) dimensional images, image regions, and other types of dense datafields (e.g., histograms). In addition, the type of each individual element in a data structure can be scalar (e.g., int, float, Boolean), complex, compound (e.g., a vector representing RGB color), and so forth.

Providing support for a combinatorial explosion of code that deals with all these data structures and types is by no means an easy task. Consequently, many existing sequential image processing libraries usually restrict support to a small set of datastructures, datatypes, and even operations [9, 45]. It is clear that such limitations have a negative effect on a library's popularity and expected lifespan.

To deal with these problems, the design and implementation of the Horus image processing library [83, 84, 85] is based on a *generic programming* approach. The Free On-line Dictionary of Computing [71] defines this approach as follows:

> Generic programming is a technique that aims to make programs more adaptable by making them more general. Generic programs often embody non-traditional kinds of polymorphism; ordinary programs are obtained from them by suitably instantiating their parameters. In contrast with normal programs, the parameters of a generic program are often quite rich in structure. For example, they may be other programs, types or type constructors or even programming paradigms.

To be more specific: given X datatypes, Y containers (data structures), and Z algorithms as essential software library components, abstraction by way of generic programming reduces the possible $X \times Y \times Z$ implementations to $X + Y + Z$ implementations. Consequently, generic programming greatly enhances library maintainability.

In Horus, generic data structures (i.e., container structures that are made independent of the type of the contained object) are implemented by way of the C++ *template mechanism* [158] — a programming concept that allows a *type* to be a parameter in the definition of a class or a function. Using the same mechanism, Horus also provides

generic algorithms that can work on the generic data structures. Because we feel that the importance of the Horus library lies more in its concepts than in its implementation, we refrain from presenting actual template code here. For more information and implementation details we refer to the Horus documentation [84, 85].

Apart from abstracting from the actual datatype each operation is applied upon, the amount of Horus library code is reduced even further by implementing only a small number of *algorithmic patterns* that covers the bulk of all commonly applied image processing operations. An algorithmic pattern corresponds to one of the *operation classes* defined in Image Algebra [131], each of which gives a generic description of a large set of operations with comparable behavior. As such, each image operation that maps onto the functionality as provided by an algorithmic pattern is implemented in Horus by instantiating the algorithmic pattern with the proper parameters, including the function to be applied to the individual data elements. As an example, an algorithmic pattern may produce a result image by applying a unary function to each pixel in a given input image. By instantiating the pattern with, for example, the absolute value operation on a single pixel, the produced output will constitute the input image with the absolute value taken for each pixel.

The version of Horus that serves as the basis for all further discussions provides the following set of algorithmic patterns:

- ***Unary pixel operation.*** Operation in which a unary function is applied to each pixel in the image. Examples: negation, absolute value, square root.
- ***Binary pixel operation.*** Operation in which a binary function is applied to each pixel in the image. Examples: addition, multiplication, threshold.
- ***Reduce operation.*** Operation in which all pixels in the image are combined to obtain a single result value. Examples: sum, product, maximum.
- ***Neighborhood operation.*** Operation in which several pixels in the neighborhood of each pixel in the image are combined. Examples: percentile, median.
- ***Generalized convolution.*** Special case of neighborhood operation. The combination of pixels in the neighborhood of each pixel is expressed in terms of two binary functions. Examples: convolution, gauss, dilation.
- ***Geometric (domain) operation.*** Operation in which the image's domain is transformed. Examples: translation, rotation, scaling.

The presented set of algorithmic patterns is not complete, as it does not cover *all* functionality required in the early stages of algorithm or application development. The Horus library, however, is subject to continuing research and extensions. Among the most important current and expected future library additions are algorithmic patterns that can be used to instantiate (1) multi pixel operations, (2) iterative and recursive neighborhood operations, and (3) queue based algorithms. Also, apart from the algorithmic pattern for geometric operations, all of the patterns that are currently incorporated in Horus are restricted to instantiating *translation invariant* operations only. Translation variant versions of the presented algorithmic patterns will be incorporated in the future as well.

3.2 Integration of Parallelism in Horus

As discussed in Section 2.3.2, the parallel library that constitutes the core of our software architecture is an extended version of the sequential Horus library. In the parallel version all additional functionality is implemented such that it does not interfere with the existing sequential code. As such, the parallel library can still be instructed to resort to traditional sequential operation — which generally is preferred over single node parallel operation due to additional overhead costs.

From a design perspective the extended library consists of four logical components, as shown in Figure 3.1. The following discusses each in turn, and identifies the relationships among them:

Component C1: Sequential Algorithmic Patterns

The first component (C1) consists of the set of sequential algorithmic patterns introduced in Section 3.1. As indicated in Figure 3.1, each algorithmic pattern present in this component is implemented as a sequence of sequential routines. All operations in such sequence must be separately available in the library — but not necessarily as user-callable routines. Apart from memory operations that may be required for the creation or destruction of internal data structures, the most important operation

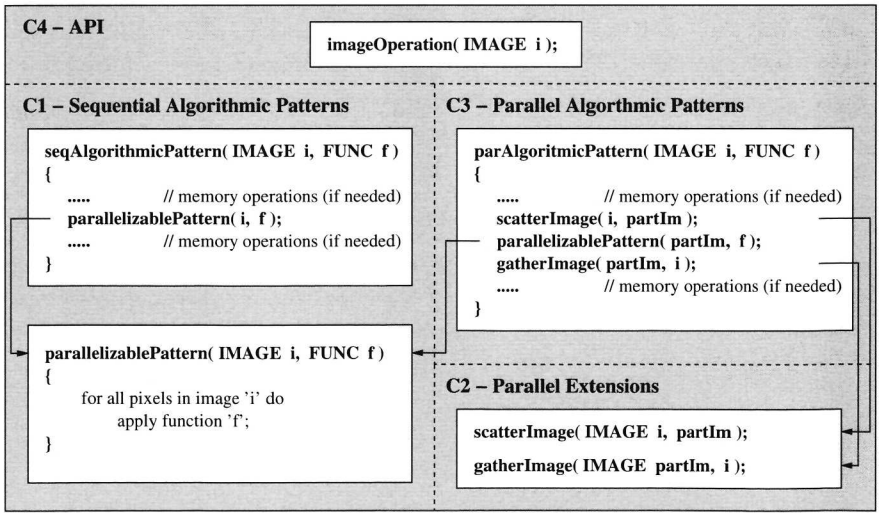


Figure 3.1: *Relationships between library components C1-C4 (note: the actual code differs substantially). Sequential code blocks that constitute a specialization of a parallelizable pattern are used in the implementation of sequential algorithmic patterns as well as in the implementation of the related parallel counterparts. All functionality is provided to the user through a sequential application programming interface (API) that contains no references to the library's parallel processing capabilities.*

in such sequence is what we refer to as a *parallelizable pattern*. Here it is sufficient to indicate that a parallelizable pattern constitutes a code block that incorporates only those instructions in a sequential algorithmic pattern that can be applied in the implementation of the related parallel algorithmic pattern as well — in the latter case without having to communicate to obtain essential data residing at any other processing unit. For a much more formal description of parallelizable patterns we refer to Section 3.4.

Component C2: Parallel Extensions

Next to the sequential algorithmic patterns, an additional set of routines is implemented to introduce parallelism into the library (component C2 in Figure 3.1). The parallel extensions deal with all aspects of parallelization, ranging from the logical partitioning of data structures to the actual exchange of data among processing units. To have full control over all interprocess communication[†], all extensions are implemented using MPI [104]. The implemented set of parallel extensions is divided into three classes:

1. Routines for data structure *partitioning*. These routines are used to specify the data structure responsibilities for each processing unit, i.e. to indicate which data parts should be processed by each node. In practice, a data structure is mapped onto a logical grid of processing units of up to 3 dimensions, which allows for optimal domain decomposition of the bulk of all image data structures (see also Chapter 5). The mapping is performed in such a way that the number of data elements each node is responsible for is well-balanced.

   The most important routines in this class are the 'doPartition()' and 'rePartition()' operations, which define the (new) responsibilities for a given data structure. Responsibilities are based on the logical grid of processing units, and the dimensionality and size of a data structure. All other routines in this class are requests for partitioning information (for example, to obtain the size and dimensionality of partial data structures other processing units are made responsible for).

2. Routines for data *distribution* and *redistribution*. These operations are used for the actual spreading (either scattering or broadcasting), gathering, and redistribution of data structures. Although the MPI 1.1 standard provides most of this functionality and the new MPI 2.0 standard defines all, we have made multiple implementations ourselves using the standard blocking MPI send and receive operations. We refer to Chapter 5 for a detailed discussion on the rationale, and the implications for application performance and optimization

   It should be noted that data distribution could have been regarded independent from data partitioning. To avoid any unnecessary communication, however, we have made the distribution of data structures dependent on the assigned data

[†]Note that in our implementations (and also in the remainder of this thesis) we assume a one-to-one relationship between processes and processing units.

responsibilities. Data partitioning is therefore always applied as part of a data distribution operation.

3. Routines for *overlap communication*. These operations are used to exchange *shadow regions* (e.g., image borders in neighborhood operations) among neighboring nodes in a logical grid of processing units.

All operations in component C2 are kernel routines, and are not made available to the library user.

Component C3: Parallel Algorithmic Patterns

To reduce code redundancy and enhance library maintainability as much as possible, much of the source code for the sequential algorithmic patterns is reused in the implementation of their respective parallel counterparts. More specifically, the implementation of each parallel algorithmic pattern is obtained by inserting communication operations from component C2 in the sequence of routines that constitutes the implementation of the related sequential algorithmic pattern. The communication routines are to obtain all non-local data (i.e. data residing on other processing units) required during execution of the parallelizable pattern. The communication routines also gather partial results data from all processing units to a single (root) node as soon as the execution of the parallelizable pattern has finished. As such, during execution all instantiations of the parallel algorithmic patterns run in a Bulk Synchronous Parallel manner [103, 162].

Component C4: Fully Sequential API

The extended image processing library is provided with an application programming interface (component C4 in Figure 3.1) identical to that of the original sequential Horus library. Due to the fact that the API contains no references to the library's parallel processing capabilities, no additional effort is required from the application programmer to obtain a parallel program. In other words: any application implemented for a sequential machine — after recompilation — can be executed on a cluster as well. As such, the library fully adheres to the first requirement of user transparency as defined in Section 2.2.

3.3 Data Parallel Image Processing

The previous sections implicitly indicated that we have adopted *data parallelism* as the programming model for implementing all parallel algorithmic patterns. In the following we clarify why we have adopted this approach as the sole technique for parallelization, rather than any other approach or even a combination of approaches. Also, to lay the foundations for the generic description of parallelizable patterns presented in Section 3.4, we give a formal description of the manner in which image data structures are represented in our data parallel library.

3.3.1 Data Parallelism versus Task Parallelism

Although many more programming paradigms for parallel computing exist, the models of *data parallelism* and *task parallelism* are used most frequently because of their effectiveness and general applicability. As defined in [50], the data parallel model focuses on *the exploitation of concurrency that derives from the application of the same operation to multiple elements of a data structure*. In other words, it is a programming model in which a single routine is applied to all elements of a data structure simultaneously. In contrast, the task parallel paradigm constitutes *a model of parallel computing in which many different operations may be executed concurrently* [170].

In the literature, a multitude of papers exists in which each of these paradigms is used effectively for parallelizing (low level) image processing operations (e.g., see [19, 32, 134, 161]). Also, for certain image processing problems it has been shown that application of a combination of the two paradigms in a single program is more effective than using either paradigm exclusively (e.g., see [112, 126]).

Despite the potential benefits of applying task parallelism, we have decided to restrict all parallel implementations in our library to the data parallel model. The reasons for this decision are as follows. First, the application of data parallelism is a *natural approach* for low level image processing, as many operations require the same function to be applied to each individual data element (or small set of elements around each data element) present in an image data structure. Second, as our parallel library is to serve as an aid in image processing *research*, the number of independent tasks available in most applications is expected to be small. This is because in the design phase of algorithms or applications, testing and evaluation generally is performed using relatively small problem sizes (e.g., using a single image rather than a database of thousands of images). A third reason is related to the *scalability* in the number of processing units. As the number of independent tasks in most image processing applications generally is much smaller than the number of elements present in the input (image) data structures, the number of processors that can be applied effectively is generally much larger in the data parallel case. Another important reason is that *load balancing* (i.e., evenly distributing all work among the available processing units) is generally much more difficult in the task parallel model. Especially in case independent parallel tasks represent highly varying workloads, it is difficult to ensure that each processor has exactly the same amount of work to do.

The decisive factor for not incorporating task parallelism in our software architecture, however, is the difficulty of combining this programming paradigm with the requirements of user transparency. The presence of a fully sequential API implies that we would have to incorporate a separate interpretation and optimization strategy to find all independent tasks available in an application. Effectively, this implies that we would have to develop, at least in part, a parallelizing compiler. For reasons explained in Section 2.2.1 we expect such compiler not to yield a desirable solution.

It would have been possible to incorporate the notion of task parallelism in the library's API, e.g. by providing aggregated operations that can work on sets of images. However, this approach would dramatically reduce the library's chances of widespread acceptance, as it would require most existing applications to be rewritten by hand

(e.g., by replacing loop constructs by a single call to an aggregated library operation). This is not trivial, as it requires the user to personally identify dependencies among tasks (which is often difficult due to the presence of indirections in the C or C++ code). To shield the user from having to deal with any of these issues, and also to avoid having to implement any optimization strategy that can detect independent tasks automatically, we have refrained from incorporating task parallelism altogether. As will be shown in Chapter 7, despite the fact that all implementations are restricted to the data parallel approach, obtained performance improvements are generally well within the efficiency requirements as put forward in Chapter 2.

3.3.2 Representation of Digital Images

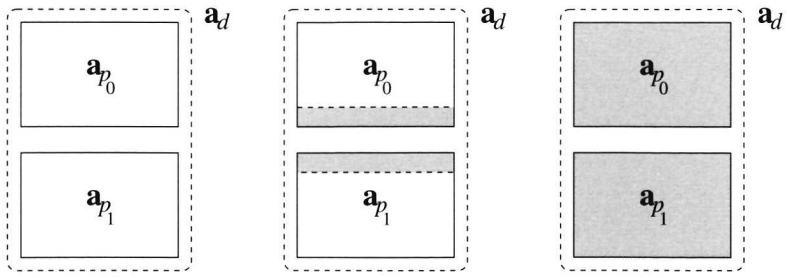
An image data structure in our library consists of a set of pixels. Associated with each pixel is a location (point) and a (pixel) value. Here, we denote an image by a lower case bold character from the beginning of the alphabet (i.e., **a**, **b**, or **c**). Locations are denoted by lower case bold characters from the end of the alphabet (i.e., **x**, **y**, or **z**). The pixel value of an image **a** at location **x** is represented by **a**(**x**).

The set of all locations is referred to as the *domain* of the image, denoted by a capital bold character (i.e., **X**, **Y**, or **Z**). Usually, the point set is a discrete n-dimensional lattice $\mathbb{Z}^n$, with $n = 1$, 2, or 3. Also, the point set is bounded in each dimension resulting in a rectangular shape for $n = 2$ and a block shape for $n = 3$. That is, for an n-dimensional image

$$\mathbf{X} = \{(x_1, x_2, \cdots, x_n) \in \mathbb{Z}^n : o_i \leq x_i \leq o_i + k_i - 1\},\ i \in \{1, 2, \cdots, n\}).$$

where $\mathbf{o} = (o_1, o_2, \cdots, o_n)$ represents the *origin* of the image, and k_i represents the extent of the domain in the i-th dimension.

The set of all pixel values **a**(**x**) is referred to as the *range* of the image, and is denoted by $\mathbb{F}$. A pixel value is a vector of m scalar values, with $m = 1$, 2 or 3. A



(a) No overlap (b) Partial overlap (c) Full overlap

Figure 3.2: *Three examples of a distributed image* $\mathbf{a}_d$ *comprising of two partial images,* $\mathbf{a}_{p_0}$ *and* $\mathbf{a}_{p_1}$. *The gray areas represent domain overlap; the white areas represent the unique domain parts.*

scalar value is represented by one of the common datatypes, such as byte, int, or float. The set of all images having range $\mathbb{F}$ and domain $\mathbf{X}$ is denoted by $\mathbb{F}^{\mathbf{X}}$. In summary, $\mathbf{a} \in \mathbb{F}^{\mathbf{X}}$ (i.e., $\mathbf{a} : \mathbf{X} \rightarrow \mathbb{F}$) is a shorthand notation for

$$\{ (\mathbf{x}, \mathbf{a}(\mathbf{x})) : \mathbf{x} \in \mathbf{X} \subset \mathbb{Z}^n \ (n = 1, 2, 3), \ \mathbf{a}(\mathbf{x}) \in \mathbb{F} \subset \{\mathbb{Z}^m, \ \mathbb{R}^m, \ \mathbb{C}\} \ (m = 1, 2, 3) \}.$$

When image data is spread throughout a parallel system, multiple data structures residing on different locations form a single logical entity. In our library, each image data structure resulting from a scatter or broadcast operation is called a *partial image*. For each partial image additional partitioning and distribution information is available. The information includes, but is not restricted to, (1) the processor grid used to map the original image data onto, (2) origin and size of the domain of the original image, and (3) the type of data distribution applied (e.g., scatter or broadcast). Partial image $\mathbf{a}$ residing on processing unit i is denoted by $\mathbf{a}_{p_i}$; its domain is denoted by $\mathbf{X}_{p_i}$. As data spreading can not result in a loss of data, for each image $\mathbf{a} \in \mathbb{F}^{\mathbf{X}}$ distributed over n processing units:

$$\bigcup_{i=0}^{n-1} \mathbf{X}_{p_i} = \mathbf{X}.$$

The n partial images related to $\mathbf{a}$ together form one logical structure, referred to as a *distributed image*. A distributed image is denoted by $\mathbf{a}_d$, and differs from a partial image in that it does not reside as a physical structure in the memory of one processing unit (unless it is formed by one partial image only). A distributed image's domain $\mathbf{X}_d$ is given by the union of the domains of its related partial images. The domains of the partial images that constitute a distributed image may be either *non-overlapping*, *partially overlapping*, or *fully overlapping* (see Figure 3.2).

Essentially, it is possible for each processing unit to perform operations on each partial image *independently*. In the library, however, we make sure that each operation (logically) is performed on distributed image data only. In all cases this results in the processing of all partial images that constitute the distributed image. This strategy is of great importance to avoid inconsistencies in distributed image data.

3.4 Parallelizable Patterns

As stated in Section 3.2, we try to enhance library maintainability by reusing as much sequential code as possible in the implementations of the parallel algorithmic patterns. To that end, for each sequential algorithmic pattern we have defined a so-called *parallelizable pattern*. Each such pattern represents the maximum amount of work in a generic algorithm that - when applied to partial image data - can be performed without the need for communication. In other words, in a parallelizable pattern all internal data accesses must refer to data *local* to the processing unit executing the operation. In the following we give a generic description of parallelizable patterns, and show their application in parallel implementations.

3.4.1 Generic Description

A **parallelizable pattern** is a sequential generic operation that takes zero or more source structures as input, and produces one destination structure as output. A pattern consists of n independent tasks, where a task specifies what data in any of the structures involved in the operation must be acquired (read), in order to update (write) the value of a *single* data point in the destination structure. In a task, read access to the source structures is unrestricted, as long as no accesses are performed outside any of the structures' domains. In contrast, read access to the destination structure in each task is limited to the single data point to be updated.

All n tasks are tied to a different *task location* $\mathbf{x}_i$, with $i \in \{1, 2, \cdots, n\}$. The set L of all task locations constitutes a subset of the positions inside the domain of one of the data structures involved in the operation (either source or destination). As a simple example, L may refer to all n pixels in an image data structure, all of which are processed in a loop of n iterations.

Each task location $\mathbf{x}_i$ has a relation to the positions accessed in all data structures involved in the operation. As such, for the parallelizable patterns relevant in image processing we define four *data access pattern types*:

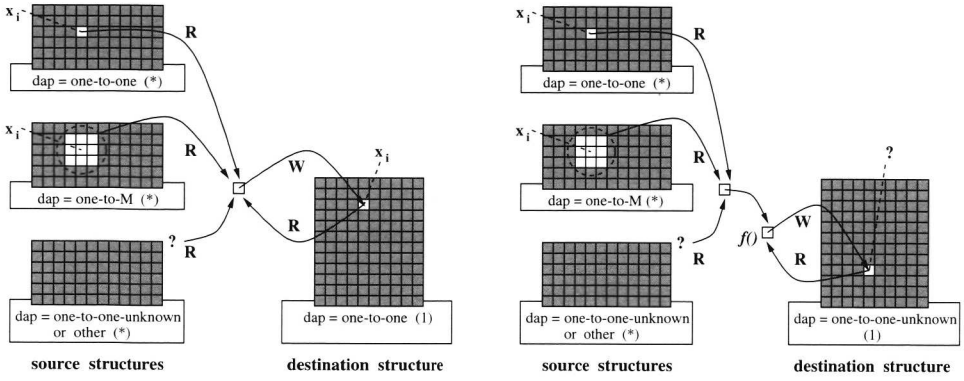
- *One-to-one.* For a given data structure, in each task T_i (with $i \in \{1, 2, \cdots, n\}$) no data point is accessed other than $\mathbf{x}_i$.
- *One-to-one-unknown.* For a given data structure, in each task T_i (with $i \in \{1, 2, \cdots, n\}$) not more than one data point is accessed. In general, this point is not equal to $\mathbf{x}_i$.
- *One-to-M.* For a given data structure, in each task T_i (with $i \in \{1, 2, \cdots, n\}$) no data points are accessed other than those within the *neighborhood* of $\mathbf{x}_i$. As an example, the 5×3 neighborhood of a point $\mathbf{x} = (x_1,\ x_2) \in \mathbf{X}$ is given by

  $$N(\mathbf{x}) = \{\mathbf{y} \in \mathbf{Y} : \mathbf{y} = (x_1 \pm j,\ x_2 \pm k),\ j \in \{0, 1, 2\},\ k \in \{0, 1\}\},$$

  where $\mathbf{X} \subset \mathbf{Y}$.
- *Other.* For a given structure, in each task either all elements are accessed, or the accesses are irregular or unknown.

A parallelizable pattern requires that for all data structures the access pattern type is given. Essentially, all four access pattern types are applicable to source structures. In contrast, the single destination structure can only have a 'one-to-one' or a 'one-to-one-unknown' access pattern type. This is because — by definition — in each task only one data point is accessed in the destination structure.

Figure 3.3 shows the two parallelizable pattern types that we discern. In a type 1 parallelizable pattern the set of task locations has a 'one-to-one' relation to the destination structure. In a type 2 parallelizable pattern the access pattern type related to the destination structure is of type 'one-to-one-unknown'. The two parallelizable patterns differ in the type of *combination operation* that is permitted. In a parallelizable



(a) Parallelizable pattern, type 1.

(b) Parallelizable pattern, type 2.

Figure 3.3: *Two parallelizable pattern types. R = read access; W = write access; dap = data access pattern; (1) = exactly one data structure of this type; (*) = zero or more data structures of this type.*

pattern of type 1 no restrictions are imposed on the combination operation. In a type 2 pattern the final combination of the intermediate result of all values read from the source structures with the value of the data point to be updated in the destination structure must be performed by a function $f()$ that is associative and commutative. Also, prior to execution of a type 2 pattern, all elements in the destination structure must have a value that is 'neutral' for operation $f()$. For example, the neutral value for addition is 0, while for multiplication it is 1.

The two parallelizable pattern types give a generalization of a large set of *sequential* image processing routines, e.g. incorporating all algorithmic patterns of Section 3.1. As such, the presented generalization captures a large majority of all operations commonly applied in image processing research (i.e., it comprises an estimated coverage of over 90%). It should be noted, however, that the two types do not present a complete coverage of the typical implementations of *all* operations in this particular field of research. For example, algorithms in which write access is to multiple data structures is required do not fall in the category of operations currently under consideration. The same holds for operations in which the value of each data point in the destination structure depends on values of other data points in the same destination structure. In how far these limitations pose any unreasonable restrictions on future library adaptations (and thus necessitates extension of the generic description of parallelizable patterns) is as of yet unknown (see also Section 3.4.5).

All algorithmic patterns that do fit into the given generalization are applicable in the process of 'parallelization by concatenation of library operations', described in Section 3.2. As discussed in the remainder of this section, on the basis of the generic description we define a standard parallelization strategy that always results in a correct data parallel implementation for any algorithmic pattern that maps onto at least one of the two parallelizable pattern types.

3.4.2 Default Parallelization Strategy

The number of elements in the set of task locations L determines the number of steps executed by a parallelizable pattern. Hence, by providing each node in a parallel system with a set $X \subset L$, the work is distributed (i.e., in a data parallel manner). In addition, the access pattern type associated with each structure involved in the operation prescribes how non-local data accesses are avoided with minimal communication overhead. As such, an optimal[‡] default parallelization strategy is obtained for any operation that maps onto one of the presented parallelizable pattern types.

First, before executing a type 1 parallelizable pattern each processing unit is provided with a non-overlapping partial destination structure that matches the elements in X. If the destination structure is updated but never read, the partial structure can be created locally. Otherwise, it is obtained by scattering the destination structure such that no overlap in the domains of the local partial structures is introduced. Before executing a type 2 parallelizable pattern, each processing unit creates a fully overlapping destination structure locally. This is always possible, as the value of all data points are given a 'neutral value', as defined by the operation.

Next, source data structures are obtained by executing (1) a non-overlapping scatter operation for each structure having a one-to-one access pattern, (2) a partially overlapping scatter operation for each structure having a one-to-M access pattern type (such that in each dimension the size of each shadow region equals half the size of the neighborhood in that dimension), and (3) a broadcast operation for all other structures. In case the values of a source structure can be calculated locally, and if it is less time-consuming to do so, no communication routines are performed at all.

Finally, when a type 1 pattern has finished, the complete destination structure is obtained by executing a gather operation. For a type 2 pattern this is achieved by executing a reduce operation across all processing units. Here, the elements that have not been updated in each local destination structure have kept a neutral value, assuring the correctness of the final reduction. In both cases, the result structure is returned either to one node, or to all.

On the basis the generic description of parallelizable patterns, the following shortly discusses parallel implementation of two example algorithmic patterns, i.e. global reduction and generalized convolution.

3.4.3 Example 1: Parallel Reduction

A sequential generic reduction operation performed on input image $\mathbf{a}$, producing a single scalar or vector value k, is defined as follows:

Let $\mathbf{a} \in \mathbb{F}^{\mathbf{X}}$, $\mathbf{x} \in \mathbf{X}$, and $k \in \mathbb{F}$, then

$$k = \Gamma\mathbf{a} = \Gamma_{\mathbf{x}}\mathbf{a}(\mathbf{x}) = \Gamma_{i=1}^{n}\mathbf{a}(x_i) = \mathbf{a}(x_1)\ \gamma\ \mathbf{a}(x_2)\ \gamma \cdots \gamma\ \mathbf{a}(x_n),$$

with γ an associative and commutative binary operation on $\mathbb{F}$.

[‡]The default strategy is optimal for operations executing *in isolation* only. In case multiple operations are executed in sequence, additional inter-operation optimization is required (see Chapter 6).

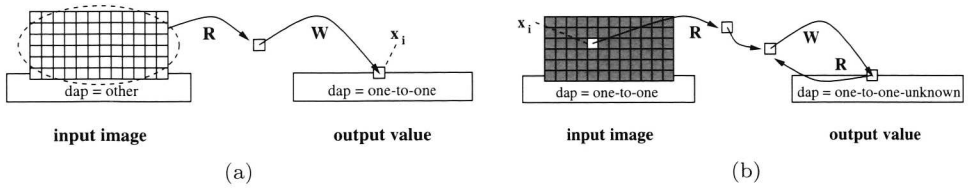


Figure 3.4: *Sequential reduction - two possible implementations.*

As shown in Figure 3.4, at least two possible sequential implementations exist for this operation. In the first implementation, the operation is performed in one step. All data points in **a** are obtained and combined to a single value, which is written out to k. In the second implementation, the operation is performed in n steps. In each step, one data point in **a** is read and combined with the current value of k.

The first implementation is a specialization of the parallelizable pattern of type 1 as described in Section 3.4; the second implementation is a specialization of the type 2 parallelizable pattern. The first implementation is not useful for our purposes, however, as its execution is limited to a single processing unit. This is because the set of task locations L consists of one element only, i.e. the location of the single output value k. The second implementation, on the other hand, is easily run in parallel as L contains all locations in input image **a**. For this implementation the input image's access pattern type is 'one-to-one'; for the single result value it is 'one-to-one-unknown'. As a result, a parallel implementation of the generic reduction operation follows directly from the generalization of Section 3.4.2. A pictorial view of the operation executed in parallel is given in Figure 3.5.

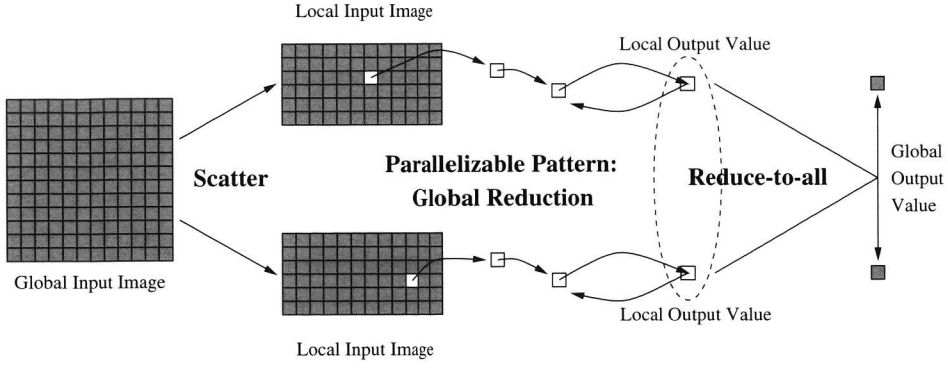


Figure 3.5: *Example reduce-to-all operation executed on 2 processing units.*

3.4.4 Example 2: Parallel Generalized Convolution

A generalized convolution performed on input image $\mathbf{a}$, producing output image $\mathbf{c}$, given a kernel $\mathbf{t}$, is defined as follows:

Let $\mathbf{a},\ \mathbf{c} \in \mathbb{F}^{\mathbf{X}},\ \mathbf{t} \in \mathbb{F}^{\mathbf{Y}},\ \mathbf{x} \in \mathbf{X},\ \mathbf{y} \in \mathbf{Y}$, with $\mathbf{X}$ having dimensionality n, and $\mathbf{Y} = \{(y_1, y_2, \cdots, y_n) : \mid y_i \mid \leq k_i \in \mathbb{Z}\}$, then

$$\mathbf{c} = \mathbf{a} \circledcirc \mathbf{t} = \{\ (\mathbf{x}, \mathbf{c}(\mathbf{x})) : \mathbf{c}(\mathbf{x}) = \Gamma_{\mathbf{y}}\ \mathbf{a}(\mathbf{x} + \mathbf{y}) \bigcirc \mathbf{t}(\mathbf{y})\ \},$$

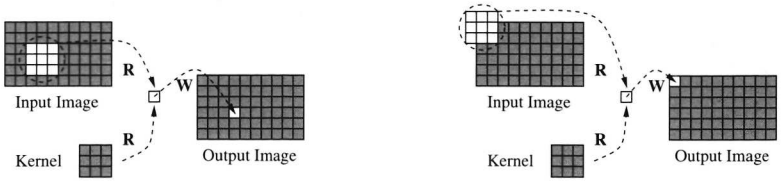
where $\bigcirc$ and γ are binary operations on $\mathbb{F}$, and γ is associative and commutative. The extent of the domain in the i-th dimension of kernel $\mathbf{t}$ is given by $2k_i + 1$. Several common generalized convolution instantiations are shown in Table 3.1.

Kernel Operation	$\bigcirc$	γ
Convolution	multiplication	addition
Dilation	addition	maximum
Erosion	addition	minimum

Table 3.1: *Example generalized convolution instantiations.*

The definition states that each pixel value in the output image depends on the pixel values in the *neighborhood* of the pixel at the same position in the input image, as well as on the values in the related kernel structure. A sequential implementation of the operation is presented in Figure 3.6. Again, set L is implicit, and contains all pixel positions in either the input image or the output image.

When comparing Figure 3.3(a) to Figure 3.6(a) it may seem that the operation directly constitutes a parallelizable pattern. Figure 3.6(b) shows that this is not the case, however, as accesses to pixels outside the input image's domain are possible. In sequential implementations of this operation it is common practice to redirect such accesses according to a predefined *border handling strategy* (e.g., mirroring or tiling). A better approach for sequential implementation, however, is to separate the border



(a) Permitted read-access (b) Forbidden read-access

Figure 3.6: *Sequential generalized convolution. Does not represent a parallelizable pattern, as read accesses outside the domain of the input image are possible (see (b)).*

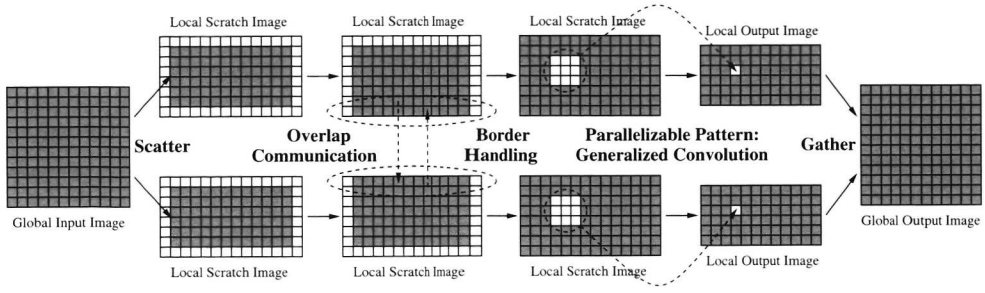


Figure 3.7: *Example kernel operation performed using 2 processing units (simplified).*

handling from the actual convolution operation. This makes implementations more robust and generally also faster, due to the fact that irregular memory accesses are avoided. For parallel implementation this strategy has the additional advantage that the algorithmic pattern for generalized convolution can be implemented such that it constitutes a parallelizable pattern.

Implementation in this manner can be performed in many different ways. In our library a so-called *scratch border* is placed around the original input image. The border is filled with pixel values according to the required border handling strategy. The newly created *scratch image* is used as input to the parallelizable pattern. Figure 3.7 depicts the operation executed in parallel. As each local scratch image has a one-to-M access pattern, an overlapping scatter of the global input image is required. In Figure 3.7 this is implemented by a non-overlapping scatter followed by overlap communication. Remaining scratch border data is obtained by local copying. Finally, the parallelizable pattern is executed, producing local result images that are gathered to obtain the complete output image. Note that Figure 3.7 gives a simplified view, as some steps of the operation are not shown. For example, depending on the type of operation, the kernel structure is either broadcast or calculated locally.

3.4.5 Discussion

The generic description of parallelizable patterns is important as it states the requirements for *sequential* implementations that are to be reused in related parallel counterparts. In addition, for each specialized parallelizable pattern implemented on the basis of the generic description, a parallelization strategy directly follows. As such, code reusability is maximized, and library maintainability and flexibility is enhanced.

It should be noted that if a sequential operation does not map onto the generic description of a parallelizable pattern, we currently take no special action to obtain good performance. In such situations, the operation is always executed using one processing unit only. In the future we will investigate whether parallelization of such operations can be generalized as well. Additional formulations may be integrated in the current generalization, or may exist independently.

3.5 Conclusions and Future Work

In this chapter we have indicated how an extensive parallel image processing library is constructed with minimal implementation effort, whilst ensuring efficiency of (sequential and parallel) execution at the same time. We have focussed on the notion of *parallelizable patterns*, and discussed how parallel implementations are easily obtained by sequential concatenation of operations that are *separately available* in the library. More specifically, on the basis of a set of four *data access pattern types*, we have obtained a default parallelization strategy for any operation that maps onto one of two parallelizable pattern types. For each image processing operation executed *in isolation*, this default parallelization strategy is optimal. This is because communication overhead is minimized, while — for the given parallelization granularity — the available parallelism is fully exploited. As such, we have shown how a parallel image processing library can be made extensible, and easily maintainable.

It is important to note, however, that in this chapter we have not discussed the important issue of inter-operation optimization, or: optimization across library calls. To obtain high performance for sequences of library routines, or for complete applications, it is not sufficient to consider parallelization and optimization of each library operation in isolation. This is because code consisting of a given sequence of parallel routines, where each routine is parallelized as described in this chapter, often contains many redundant communication steps. Also, it is often possible to further reduce communication overhead by combining multiple messages in a single transfer. Our solution to this fundamental problem, and the integration of this solution in our software architecture, is discussed extensively in Chapters 4, 5, and 6 of this thesis.

In the future the generic description of parallelizable patterns may need to be extended or adapted to capture image library additions and extensions. For example, at the time of writing it is not entirely clear whether the optimal parallel implementation of recursive filter operations as described in [28, 40] can be derived from one of the presented parallelizable pattern types. As a consequence, we may need to investigate for what type of image operations the strategy of 'parallelization by concatenation of library routines' breaks down, i.e., does not provide efficient implementations, or can not be applied at all. Still, the presented description of parallelizable patterns is important, as it prescribes the *sequential* implementation of a large majority (i.e., over 90%) of all operations commonly applied in image processing research. Any implementation obtained in this manner can be applied *without change* in efficient parallel implementations as well — thus avoiding unnecessary code redundancy, and minimizing the required implementation effort.

Chapter 4

Semi-Empirical Modeling of Parallel Low Level Image Processing Operations*

"Pour avoir une vérité il faut deux facteurs — un fait et une abstraction."

Remy de Gourmont (1858 - 1915)

As described in Chapter 3, for each sequential algorithmic pattern available in our library we have implemented *only one* parallel counterpart. Because no single parallel implementation is guaranteed to provide optimal performance on all target platforms, each operation is implemented such that it is capable of adapting to the specific performance characteristics of a parallel machine at hand. In the previous chapter we have indicated that two of the parameters that determine an application's parallel performance are fixed in the library implementations: i.e., the parallelization *granularity* (or, the amount of computation performed between two communication steps) as well as the data dependencies. Optimization decisions relating to several additional parameters, however, are still made at application compile time, and even at run time. Such parameters include (1) the logical processor grid to map data structures onto, (2) the routing pattern for the distribution of data, (3) the number of processing units, and (4) the type of data distribution (e.g., broadcast instead of scatter).

To make optimization decisions automatically, knowledge is required of the performance characteristics of the routines applied in a particular application. In our software architecture we have incorporated this knowledge by annotating each user-callable library operation with a performance model for run time cost estimation. Due

*This chapter combines our papers published in *Proceedings of the 7th International Euro-Par Conference (Euro-Par 2001)* [147] and *Parallel Computing* [149].

to the intended portability of the software architecture to clusters, the performance models have been designed to be applicable to *all* machines in this class of architectures. Also, the complexity of the models is kept to a minimum to allow high-speed evaluation of complete applications possibly even at run time. In addition, the models are capable of generating estimations of sufficiently high accuracy to allow optimization decisions to be made correctly.

In the literature a multitude of performance analysis and modeling techniques has been described. The techniques range from direct measurement to detailed mathematical and simulation models, and adhere to widely varying performance requirements in terms of both estimation accuracy and speed of evaluation. As will be discussed in this chapter, a major problem with existing performance estimation techniques is that these generally incorporate a direct relationship between the estimation accuracy and the technique's complexity (for example, the number of model parameters). In other words, increased estimation accuracy is obtained at the expense of greater complexity and reduced evaluation efficiency.

In this chapter we propose a *semi-empirical modeling* technique that is specifically designed to overcome this problem. While being simple and portable, the semi-empirical modeling approach also provides a sufficiently high estimation accuracy. The approach is based on a high-level abstract machine definition (the Abstract Parallel Image Processing Machine, or APIPM) which is designed to capture typical behavior of low level image processing operations executing on a cluster. From the APIPM *instruction set* a high-level abstract performance model is obtained that is applicable to all such platforms. The crux of the semi-empirical modeling approach is that an additional *benchmarking* phase is required to capture implicit but essential cost factors, and to bind each abstract model parameter to a concrete performance estimation for a parallel machine at hand.

Hence, the primary research issue addressed in this chapter is as follows: How to apply benchmarking in combination with simple analytical models to obtain accurate performance estimates for optimization of complete parallel image processing applications? In this respect, it is interesting to note that this research issue closely relates to the more general problem statement put forward by Professor Tony Hey in his invited talk at the Euro-Par 2001 conference: "The ultimate goal in the field of parallel and distributed computing is to use a combination of benchmarking kernels and simple models for accurate performance estimation of full applications" [68]. Essentially, our APIPM-based semi-empirical modeling approach forms a domain-specific solution to this much broader — and as of yet: unsolved — problem.

This chapter is organized as follows. Section 4.1 investigates the requirements for a performance estimation technique to be applied in our software architecture. Several existing approaches are evaluated according to these requirements as well. On the basis of two estimation techniques described in the literature, a generalized description of our semi-empirical modeling approach is given in Section 4.2. Section 4.3 introduces the APIPM and its instruction set. The APIPM-based performance models, and the applied benchmarking technique, are presented in Section 4.4. In Section 4.5 model predictions are compared with results obtained on a real machine from the class of platforms under consideration. Concluding remarks are given in Section 4.6.

4.1 Computer System Performance Estimation

The success of our software architecture greatly depends on the quality of the code optimization, which is to be performed automatically, hidden from the user. Code optimization is implemented by leaving each operation in the library a choice between several parallelization strategies. Each such strategy has a different effect on the performance on each parallel platform. By providing accurate estimations of the performance of each strategy for the parallel machine at hand, the fastest code alternative is selected with ease.

The effectiveness of the optimization process entirely depends on the technique for estimating the performance of a computer system. In the following we present the general requirements for a performance estimation technique to be applied in our software architecture. In the light of these requirements a short evaluation is given of the most significant estimation approaches described in the literature.

4.1.1 Estimation Technique: Requirements

A performance estimation technique designed for our purposes should incorporate all relevant tasks typically performed by data parallel imaging operations. In our case such tasks relate to either *computation*, *communication*, or *I/O*. Computational tasks include all parallelizable patterns as defined in Chapter 3, as well as the basic memory operations for creation, destruction, and copying of data structures. Communication tasks are formed by the bulk of operations from the set of parallel extensions described in the previous chapter, including overlap communication and all distribution and redistribution routines. I/O tasks include all operations for transporting data between a processor's main memory and external devices such as disk drives and cameras.

Apart from having to reflect the typical behavior of parallel low level image processing routines, the performance estimation technique should also conform to the following (more general) requirements (similar to [62]):

1. *Simplicity.* In a realistic estimate, the number of samples is proportional to the number of parameters. To reduce the costs of performance evaluation, the number of free parameters should be kept to a minimum.

2. *Accuracy.* To make sure the architecture can make correct optimizations, the generated performance estimations must be of sufficiently high accuracy. The degree of accuracy is considered sufficient if correct decisions are made in at least 95 percent of all cases, and poor decisions are generally avoided.

3. *Applicability.* For portability, the performance evaluation technique integrated in our software architecture must be applicable to all clusters.

It is important to note that — in general — the requirement of simplicity enhances applicability, but reduces accuracy. Therefore, care must be taken in the design of the estimation technique to ensure that it can produce good performance estimates with relative ease.

4.1.2 Estimation Techniques in the Literature

Techniques for computer system performance estimation abound in the literature. Roughly speaking, each such technique can be classified into one of three main categories: (1) *measurement*, (2) *modeling* and (3) *hybrid* methods. Estimation techniques that belong to the second category can be further divided into the subcategories of (2a) *mathematical analysis* and (2b) *simulation* [72].

Category 1: Measurement

Performance estimation by *measurement* is generally performed on a real system under conditions that reflect typical workload and behavior. Execution times of real problems are then inferred from measured results. Application of this approach in our software architecture has several drawbacks. First, in many cases the complete system to be evaluated has yet to be developed, and may change over time. Second, even if a complete system is available it is often not clear what workload is realistic or typical. Finally, if the measurement process is biased towards certain aspects of the underlying hardware, the measurement technique may not be applicable to other platforms.

Benchmarking is an alternative technique, which is often used for comparison of multiple computer systems (e.g., see [39, 69, 167]). Rather than reflecting typical behavior, benchmarks often represent non-typical, artificial workloads. In comparison with direct measurement, benchmarking has the advantage that the system to be evaluated does not have to be available. The use of non-typical workloads, however, often has a negative effect on the accuracy of the performance estimations. A solution — albeit complex — is to capture results for small instruction mixes and a variety of workloads, and to interpret the measurement results with utmost care [44, 156].

Category 2: Modeling

Performance *modeling* can be applied in cases where direct measurement is too costly, or where the computer system to be evaluated is not available. In the category of *mathematical analysis*, models range from simple (linear) algebraic expressions to complex formalisms such as queueing networks [72, 135]. In general, such models have a high response time due to their ease of evaluation. An additional advantage is that parameter values may be varied to observe their relative impact on performance. However, to obtain high estimation accuracy, the large number of model parameters may violate the simplicity and applicability constraints.

In *simulation models* behavior and workloads are described (imitated) in a special computer program — usually an annotated or otherwise adapted version of a 'real' program [72, 123]. Performance predictions are obtained by monitoring the execution of the adapted program. The main advantage of simulation models is that dynamic system behavior is easily captured. Also, simulation makes it easy to 'zoom in' on interesting or expensive parts of a system. A disadvantage is that the system to be evaluated must be available, at least in some rudimentary form. Another drawback is that it is a costly method for obtaining even moderately accurate performance estimates, thus violating the simplicity constraint.

Category 3: Hybrid Methods

In hybrid estimation techniques a combination of measurement and modeling is applied [108, 172]. Such techniques have the advantage that the complexity of using either measurement or modeling in isolation can be avoided, while a high level of estimation accuracy can still be obtained. The following discusses two hybrid approaches that form the basis for the estimation technique applied in our software architecture.

1. Machine Characterization Based on an Abstract Fortran Machine

In [133], Saavedra-Barrera et al. acknowledge that many state-of-the-art sequential computer systems have become too complex to be accurately captured in a mathematical model. The authors measure system performance in terms of an Abstract Fortran Machine (AFM) — an approach referred to as *narrow spectrum benchmarking*. The AFM instruction set consists of the primitive operations available in Fortran, such as arithmetic and logical functions, procedure calls, loops, etcetera. All primitive operations are measured separately, and the combined set of measurements characterizes a specific machine. The approach is based on the assumption that the execution time of any program can be partitioned into independent time intervals, each corresponding to one AFM instruction. Although, in general, high level operations are never completely independent (e.g., due to compiler optimizations), the authors have shown that the assumption is reasonably accurate for a wide range of systems [132]. It should be noted that an earlier technique, described by Peuto et al. [122], is similar to the AFM-based approach. It is different, however, in that all machine characterizations are incorporated at the much lower level of *machine instructions*.

The model of the total execution time of a program **A** as described in [133] is formalized as follows. Let $\mathbf{P_M} = \langle P_1, P_2, \cdots, P_n \rangle$ be the set of parameters that characterizes the performance of machine **M**. Each of the n performance parameters is related to a different operation in the AFM instruction set. Let $\mathbf{C_A} = \langle C_1, C_2, \cdots, C_n \rangle$ be the normalized dynamic distribution of the AFM instructions present in program **A**, and let C_{total} denote the total number of AFM instructions executed in program **A**. The expected execution time of program **A** on machine **M** is then obtained by

$$T_{A,M} = C_{total} \sum_{i=1}^{n} C_i P_i = C_{total} \mathbf{C_A} \cdot \mathbf{P_M}, \quad with \sum_{i=1}^{n} C_i = 1.$$

The authors indicate that the only way in which this linear model can give accurate results is when (1) the measurements of the AFM instructions are representative of typical occurrences in real programs, (2) errors caused by the intrusiveness of the measurements are not significant, and (3) variance in the mean execution time caused by the system, and by the instructions themselves, is small. Still, experiments have shown that for many applications the performance predictions were sufficiently close to actual execution times. In general, occurrences of bad estimations were easily explained by code optimizations performed by the compiler, which had not been captured in the benchmarking process.

2. Incorporating System Variance by Adaptive Sampling

The AFM-based approach of narrow spectrum benchmarking provides a solution to the problem of the high complexity of complete analytical study of computer systems. A drawback of the approach, however, is that system variance is almost completely ignored. For applications working on extensive dense data fields (e.g., image data structures) this is too crude a restriction as variations in the hit ratio of caches and system interrupts often have a significant impact on performance [59, 136].

In [90] a prediction method is presented that incorporates both program behavior and machine variance. The predictions are based on the approach of *adaptive sampling*, which is constrained by a fixed time budget for all measurements. In other estimation methods significant inaccuracies in performance estimates may arise, as a known execution time for one input size is often a poor predictor of the performance for other input sizes. In general, the main source of variation is due to the availability of small amounts of fast cache memory. As there is a decreasing portion of data residing in cache with increasing input size, linearity in response is disturbed.

This problem is attacked by the adaptive sampling approach, which measures the execution time of an algorithm for several input sizes. The advantage of the approach is that, in part, it also incorporates sources of variation inherent to an application. In matrix multiplication, for example, a linear increase in the sizes of the data-structures being applied results in a non-linear growth in execution time. Another nice feature of the approach is that it fixes the time needed for the measurement process. One may be tempted to run a benchmark at the largest size believed to fit within the budget. However, due to the many possible sources of variation the assumed execution time may be far from realistic.

Some image processing functions (e.g., data-driven segmentation) have an inherent randomness, and an execution time that is much less predictable. For such algorithms it is difficult to obtain accurate estimations on the basis of adaptive sampling. Another problem is that the approach may only measure small sized inputs not representative of typical workloads. As will be discussed in Section 4.4, due to these latter two problems we have chosen to apply a measurement technique similar, but not identical, to the adaptive sampling approach.

A Combination of Techniques

From the presented overview we conclude that several effective estimation techniques exist that are based on measurement, modeling, or a combination thereof. Unfortunately, no estimation technique exists that provides a level of abstraction that is truly applicable for optimization of applications implemented using our software architecture. Also, many measurement techniques have proven to be a weak basis for accurate performance estimation, as the impact of *system variance* is often ignored [90].

In the following section we present a description of the performance estimation approach applied in our software architecture. Essentially, it is a combination of the two hybrid techniques described above, as it integrates the impact of system variance with high level abstractions relevant for image processing applications. We refer to our approach as *semi-empirical modeling*.

4.2 Semi-Empirical Modeling

As stated, any accurate performance estimation technique should cover all relevant aspects of a computer system under consideration. Consequently, many existing estimation techniques incorporate detailed behavioral abstractions relating to the major components of such a system [72, 100], a.o. including: (1) the central processing unit, (2) the memory hierarchy, including multiple cache levels, (3) I/O devices, (4) the interconnection network, (5) the operating system, and sometimes also (6) a specific piece of application software. A major problem with this approach is that one may need tens, if not hundreds, of platform-specific machine abstractions to obtain truly accurate estimations. Consequently, the essential requirements of *simplicity* and *applicability* as put forward in Section 4.1.1 are not satisfied.

To overcome this problem we have designed a new technique for performance estimation of parallel image processing applications running on clusters. The technique, which we refer to as *semi-empirical modeling*, allows for high-speed evaluation of complete applications or any relevant constituent subtask. Also, the technique is sufficiently accurate to allow correct optimization decisions to be made automatically, on any machine in the class of target platforms. The semi-empirical modeling approach is based on three essential ingredients:

1. A high level abstract machine definition for parallel low level image processing (the APIPM), including a related instruction set.
2. A simple, APIPM-based, linear performance model related to each user-callable library operation.
3. A benchmarking method — aimed at the application domain of low level image proceessing — to capture essential cost factors not made explicit in the models.

In other words, the technique is based on a combination of relevant *abstraction*, simple *modeling*, and domain-specific *measurement*.

The essence of the semi-empirical modeling approach is that any behavior or cost factor that can not be assumed identical for all target platforms is abstracted from in the definition of the model parameters. To still bind each abstract model parameter to an accurate performance estimation for a parallel machine at hand, benchmarking is performed on a small set of sample data to capture all such essential, but implicit cost factors. In the remainder of this chapter the three essential ingredients of our modeling approach are discussed in more extensive detail.

4.3 Abstract Parallel Image Processing Machine

As in the AFM-based approach described in Section 4.1.2, we have introduced well-defined system abstractions by specifying a high level abstract machine for image processing: the Abstract Parallel Image Processing Machine, or APIPM. In the APIPM common hardware characteristics of the target machines are reflected by the definition of *abstract hardware components*. In addition, the typical behavior of the routines to be run are reflected in a related *instruction set*.

4.3.1 APIPM Components

An APIPM consists of one or more identical abstract sequential image processing machines (ASIPMs), each consisting of four closely related components (see Figure 4.1):

1. a *sequential image processing unit (SIPU)*, capable of executing APIPM instructions, one at a time,
2. a *memory unit*, capable of storing (image) data structures,
3. an *I/O unit*, for transporting data between the memory unit and external sensing or storage devices,
4. *data channels*, the means by which data is transported between the ASIPM units and external devices.

Although the memory unit of each ASIPM is connected with those of all other ASIPMs, no ASIPM has direct access to data maintained by any other ASIPM. The ASIPMs are ordered and identified by a unique number. The range of valid identifiers is $0, ..., N-1$, where N is the number of ASIPMs in the APIPM. Each ASIPM has knowledge of the range of valid identifiers, and of its own unique number.

The definition of the APIPM reflects a state-of-the-art homogeneous commodity cluster. It only differs from a general purpose machine in that each sequential unit is

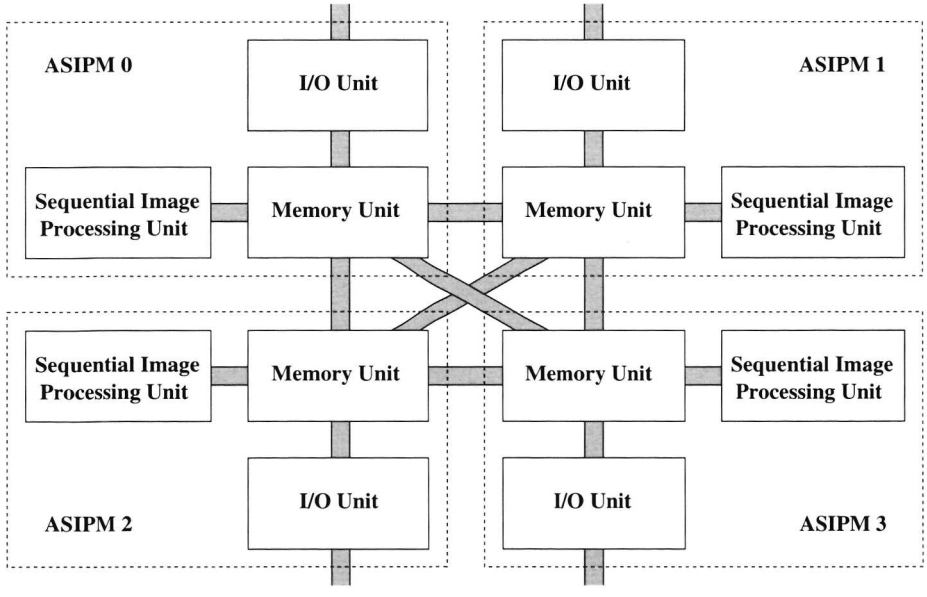


Figure 4.1: *Abstract Parallel Image Processing Machine (APIPM) comprising of four ASIPMs.*

designed for image processing related tasks only. Each ASIPM is capable of running code individually, independent of all other ASIPMs. The programs executed by each ASIPM need not be identical. Exchange of data between processing units is possible by communication over the interconnecting data channels.

Although most realistic clusters do not have a fully connected communication network, we have decided to include one in the APIPM. This is because in modern multicomputer systems data transfer between nodes that are not directly connected does not require the intermediate nodes on the complete send path to be interrupted. Consequently, the time required for transferring a message from one node to another is not significantly influenced by the distance between the nodes.

4.3.2 APIPM Instruction Set

The APIPM instruction set (see Table 4.1) consists of four classes of operations:

1. *Generic image processing instructions*, i.e. the specialized parallelizable patterns described in Chapter 3.
2. *Memory instructions*, for allocation and copying of (image) data.
3. *I/O instructions*, for transporting data between the memory unit and external devices.
4. *Communication instructions*, for exchanging data among ASPIM units.

For reasons of simplicity, in the overview of Table 4.1 the operands (i.e., arguments) for each opcode have been left out. A complete overview will be given in Section 4.A.

In the instruction set we have included only two communication instructions (i.e., `SEND` and `RECV`). Collective communication operations are not included, as these can be implemented using the two point-to-point operations. The definition of the `SEND` and `RECV` instructions is identical to the standard blocking communication operations available in MPI [104] (i.e., `MPI_Send()` and `MPI_Recv()`).

In the abstract machine multiple real-world objects must be represented, which should be passed as parameters to the APIPM instructions. The most prominent objects are images, but templates, matrices, and the likes, are essential as well. In the instruction set we do not introduce a special data representation for each of these objects. As will be explained in detail in Section 4.A, we make use of *memory references* instead. Such references contain information about the internal data representation, but lack any semantic information. The semantics are determined by the APIPM instruction the memory reference is passed to as a parameter.

It is important to note that for several generic image processing operations in the instruction set *data element homogeneity* is required. This means that the scalar type and the dimensionality of the elements in multiple data structures passed as parameters to a single instruction must be identical. The restriction of data element homogeneity is enforced to acknowledge the differences between operations on homogeneous and heterogeneous types. If homogeneity would not be required additional casting or copying of data would be hidden inside the APIPM. For many instructions such additional tasks constitute a significant overhead, which must be made explicit.

opcode	generic image processing instructions
UPOP	unary pixel operation
BPOPV	binary pixel operation (constant value as argument)
BPOPI	binary pixel operation (complete image as argument)
REDUCOP	global reduction operation
NEIGHOP	neighborhood operation
GCONVOP	generalized convolution
GEOMAT	geometric transformation (matrix as argument)
GEOROI	geometric transformation (region of interest)
opcode	memory instructions
CREATE	allocate data block in memory unit
MEMCOPY	copy data in memory unit
DELETE	free up data block in memory unit
opcode	I/O instructions
IMPORT	import data from external device
EXPORT	export data to external device
opcode	communication instructions
SEND	send data to other ASIPM
RECV	receive data from other ASIPM

Table 4.1: *Simplified APIPM instruction set.*

4.3.3 Discussion

The definition of the abstract parallel image processing machine and its related instruction set is not complete, as it can not be used as a basis for an actual implementation. This, however, is also not the reason for introducing the abstract machine. We stress that the APIPM is defined to serve as a basis for platform independent performance models. Many components deliberately have been left out of the specification, to keep the APIPM-based performance models as simple as possible.

The specification includes no information on how to load programs on each ASIPM unit. Also, no memory area has been identified in which APIPM programs are stored, and no program table or program counter has been defined. In other words, all hardware components that are essential to actually let programs run on the APIPM are left out of the specification. All such components are deemed too low level to be of any use in the performance models, and hence are not incorporated in the APIPM.

The APIPM instruction set is not complete either. For example, the APIPM lacks value comparison and conditional jump instructions. Such instructions have a relatively insignificant impact on execution time, and should not be incorporated in a performance model. Also, no instructions are included to set up special data structures, such as templates, and matrices. Again, such instructions are essential for the APIPM to run correctly, but the effect on the execution time in general is insignificant. In this respect, one may argue that the "DELETE" operation is expected to have no effect on performance either, and should have been left out of the instruction

set as well. However, this instruction is essential to see which memory references are still in use, and which are not. As will be discussed in Chapter 6, this knowledge is of great importance in the optimization and scheduling of complete applications.

4.3.4 Related Work

In the field of parallel low level image processing the definition and use of abstract machines is relatively new. In fact, the only references we know of are all from one research group at *The Queen's University of Belfast* in Northern Ireland. In several papers Crookes et al. [35, 36] discuss the design of a *Portable Parallel Abstract Machine (PPAM)*, whose instruction set is based on the *Image Algebra Language (IAL)*, which in turn is based on Image Algebra [131]. As discussed in Section 2.2.2, IAL is a machine independent programming language capable of parallel implementation on a range of distributed memory parallel machines. In later work both the PPAM and IAL have been extended considerably. The languages *I-BOL* [20] and *TULIP* [155] are two of the more sophisticated extensions of IAL. In later papers, a more recent version of the PPAM is referred to as the *EPIC abstract machine* [34]. The basic ideas behind the abstract machines have not changed throughout the years, so in the remainder of this discussion we will only consider the original PPAM.

The Portable Parallel Abstract Machine is designed as the hypothetical target machine for the IAL compiler. The PPAM consists of two main components: a sequential controller (implemented on a front end machine, such as a SUN workstation), which communicates with an abstract parallel co-processor (see Figure 4.2). This co-processor can be any kind of parallel system. The use of the parallel co-processor by the sequential controller can be thought of in rather the same way as a floating point co-processor is used by a microprocessor. Although the PPAM design is dissimilar to that of the APIPM, its related instruction set is almost identical to ours.

The main differences from our work stem from the fact that the PPAM is used as an aid in the design of a parallel compiler rather than as a basis for a performance model definition. On the one hand, the PPAM incorporates a higher level of abstraction than the APIPM, as the communication aspects of parallel execution are not incorporated in its definition. On the other hand, the inclusion of low level abstract hardware components (such as an instruction control unit) often makes the abstraction level much lower. Essentially, the differences in the design of the two abstract machines are explained by the fact that the two research directions are non-overlapping.

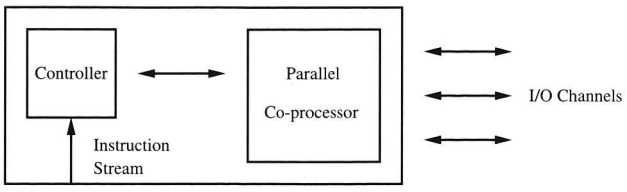


Figure 4.2: *Portable Parallel Abstract Machine [36].*

4.4 APIPM-Based Performance Models

In our software architecture, all library operations are assumed to be implemented by concatenation of APIPM instructions only. Also, it is assumed that the execution time of each library operation can be partitioned into *independent* time intervals, each corresponding to the cost of a single APIPM instruction. The performance of a library operation is then simply obtained by adding the execution times of all APIPM instructions used.

Similar to the AFM-based models described in Section 4.1.2, this idea is formalized as follows. Let $\mathbf{I} = \{I_1, I_2, \cdots, I_n\}$ be the APIPM instruction set. Also, let $\mathbf{P} = \{P_{I_1}, P_{I_2}, \cdots, P_{I_n}\}$ be the set of *performance values* for all n instructions in $\mathbf{I}$. We assume that, for any given system capable of running APIPM instructions, and for each instruction in $\mathbf{I}$, P_{I_i} can be obtained by benchmarking. Also, let $\mathbf{L} = \{\mathbf{L}_1, \mathbf{L}_2, \cdots, \mathbf{L}_m\}$ be the set of all m operations implemented using instructions in $\mathbf{I}$ only. For all library operations $\mathbf{L}_x$ ($x \in \{1, \cdots, m\}$) we define $\mathbf{L}_x = \{I_1, I_2, \cdots, I_n\}$, in combination with the total number of occurrences (or *count*) of each APIPM instruction in $\mathbf{L}_x$: $\mathbf{C}_x = \{C_{I_1,x}, C_{I_2,x}, \cdots, C_{I_n,x}\}$. The count of each instruction can have any value in $\mathbb{N}$ (including 0). The expected total execution time of each library operation $\mathbf{L}_x$ is then obtained by

$$T_{\mathbf{L}_x} = \sum_{i=1}^{n} C_{I_i,x} P_{I_i}.$$

Similarly, the expected total execution time of any application implemented using our library is obtained by adding the execution times of each library operation used.

A problem with the simplistic model formalized here is that most APIPM instructions are not single static entities. This is because the execution of an instruction is often dependent on the values of its operands. Therefore, a static entity for each possible operand combination must be incorporated in our model. To avoid an explosion of the number of static entities we allow each instruction I_i and each value P_{I_i} to be *parameterized*. Because the operands of the APIPM instructions are discussed in the appendix to this chapter (Section 4.A), a detailed overview of the model parameterization is deferred to the appendix as well. To give a straightforward example, however, in almost all APIPM instructions a 'datatype' parameter is incorporated (e.g., giving $I_i('int')$ and $I_i('float')$). Also, a 'data-input-size' parameter is required for most performance values in $\mathbf{P}$ (e.g., giving $P_{I_i(datatype)}(size)$). The choice of model parameters is dependent on the actual implementation of each APIPM instruction. For more detailed information we refer to Section 4.B.

Benchmarking

To capture system variation without having to rely on platform specific model parameters, the semi-empirical modeling approach requires an additional benchmarking phase to be performed. For our software architecture to be used on a specific platform, benchmarking results need to be obtained *only once*. As long as the underlying hardware layers and supporting software layers (e.g., operating system, compiler, etcetera)

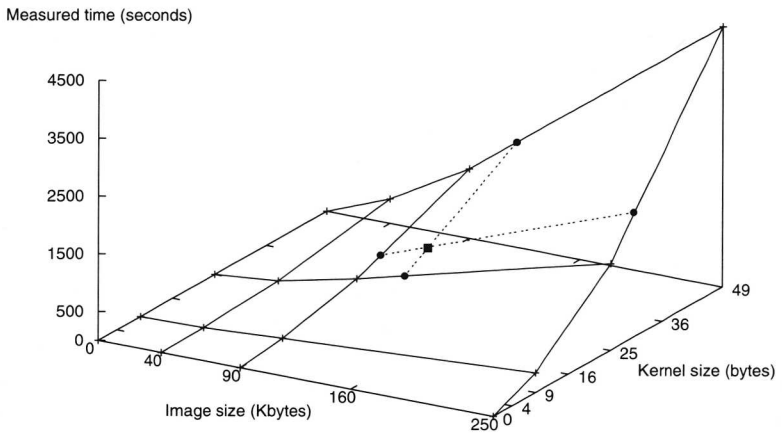


Figure 4.3: *Performance estimation in case of neighborhood operations and generalized convolutions, whose performance values depend on two input size parameters — i.e., image size and kernel size. Any required performance value (above represented by the small black square) is obtained by simple bilinear interpolation. In this example, measurements were performed for image sizes of 40, 90, and 250 Kb, each combined with kernel sizes of 9, 25, and 49 bytes.*

are not upgraded, the same set of measurement results can be applied for estimation of *any* application implemented using our parallel image library.

As in the adaptive sampling approach of Section 4.1.2, in our software architecture each APIPM instruction is measured for *multiple* input sizes. In contrast to adaptive sampling, however, we do not define a fixed time budget for all measurements. By default we use a small, predefined set of input sizes for all benchmarking operations. To avoid the benchmarking phase to be unacceptably lengthy, the set of input-sizes may be user-defined as well.

To capture non-linear performance growth without having to perform measurements for *any* possible workload, between each pair of measured performance values performance growth is taken to be piecewise linear. For estimation of instructions whose performance value is dependent on one data input size parameter this interpolation is straightforward. The performance values of neighborhood operations and generalized convolution operations, however, are dependent on two data input size parameters - i.e., the size of the input image and the size of the kernel or template structure. As is indicated in Figure 4.3, in such situation we apply bilinear interpolation to obtain the required performance estimation.

4.4.1 Extended Model for Point-to-Point Communication

Whereas the performance model described above is sufficient for all sequential APIPM instructions, an extension is required for the two communication operations (i.e., `SEND` and `RECV`). First, this is because an accurate prediction of the *end-to-end communication time* usually can not be obtained by considering the time a processor is busy executing a `SEND` or a `RECV` instruction alone. Second, in its current form the model does not closely match the capabilities of the communication instructions as defined in MPI. Most notably, the impact of a message's memory layout on communication costs is not incorporated in the model. This is an important point, as one of the tasks of the scheduler of Section 2.3.2 is to make decisions regarding the domain decomposition of an application under consideration. Depending on the type of such domain decomposition, it may be necessary to communicate data stored noncontiguously in memory. As was shown by Prieto et al. [125], knowledge of a message's memory layout is important, as non-unit-stride memory access may have a severe impact on performance due to caching. Also, the MPI operations may handle the transmission of noncontiguous data differently from contiguous blocks, possibly causing additional overheads due to the packing of data into a contiguous buffer.

To incorporate such essential cost factors we have designed an extended model for point-to-point communication. The model, called P-3PC (or the *Parameterized model based on the Three Paths of Communication*), closely resembles other models described in the literature (e.g., the Postal Model [11, 21], LogP [38], and LogGP [1]). The model is capable of modeling the essential communication patterns as used in data parallel image processing applications. In addition, and in contrast to the models mentioned above, it is also capable of accurately predicting the communication costs related to any type of domain decomposition. As this topic is outside the scope of the current chapter, an extensive overview of the P-3PC model and its capabilities is given in Chapter 5. In the evaluation of the APIPM-based models presented in the remainder of this chapter, all P-3PC specific modeling properties have been left out.

4.4.2 Discussion

The most important advantage of the APIPM-based performance models is that predictions are based on very *high level* instructions — even in comparison with the AFM-based models of Section 4.1.2. It would have been possible to define a model on the basis of much lower level instructions as well, but execution times of such instructions tend to be less independent than those of higher level instructions. This is mainly due to optimizations performed by the applied compiler. Also, it is much more difficult to obtain accurate values for lower level instructions, due to the inherent intrusiveness of the benchmarking process

A possible drawback of the models is that the instructions and related performance values are parameterized with quite a large number of instruction behavior and workload indicators. Obtaining accurate performance values for all possible combinations of parameters is both costly and difficult. However, it is possible to combine several parameters to obtain a more general indicator. As an example, promising candi-

dates for parameter merging are those that relate to data structure sizes (e.g., width, height, depth, etc.). Furthermore, the benchmarking tool that plays an essential role in our software architecture is implemented such that it allows a user to set regions of interest, to restrict the set of all possible measurements. In addition, it is possible to let the benchmarking process be run in parallel on multiple nodes within a target architecture, to reduce the benchmarking costs even further.

4.5 Measurements and Validation

In this section we show how a realistic image processing application, implemented using our software architecture, is executed in parallel. The application is highly relevant as it incorporates all of the important APIPM instructions defined in Section 4.3.2. First, a short description is given of the underlying algorithm. Next, both a straightforward sequential implementation as well as its related parallel execution are discussed. Finally, measured results are compared with APIPM model predictions.

4.5.1 Detection of Curvilinear Structures

As discussed in [55, 56], the problem of detecting curvilinear structures in images is solved by considering the second order directional derivative in the gradient direction, for each possible line direction. This is achieved by applying anisotropic Gaussian filters, parameterized by orientation θ, smoothing scale σ_v in the line direction, and differentiation scale σ_w perpendicular to the line (Figure 4.4), given by

$$r''(x, y, \sigma_v, \sigma_w, \theta) = \sigma_v \sigma_w \left| f_{ww}^{\sigma_v, \sigma_w, \theta} \right| \frac{1}{b^{\sigma_v, \sigma_w, \theta}}. \tag{4.1}$$

When the filter is correctly aligned with a line in the image, and σ_v, σ_w are optimally tuned to capture the line, filter response is maximal. Hence, the per pixel maximum line contrast over the filter parameters yields line detection:

$$R(x, y) = \arg \max_{\sigma_v, \sigma_w, \theta} r''(x, y, \sigma_v, \sigma_w, \theta). \tag{4.2}$$

This directional filtering problem can be implemented sequentially in many different ways. For each orientation θ it is possible to create a new filter based on σ_w and σ_v. In effect, this yields a rotation of the filters, while the orientation of the input image remains fixed. Another possibility is to keep the orientation of the filters fixed, and to rotate the input image instead. Yet another solution is to integrate the notion of orientation in the filter operation itself. In this case image pixels are accessed according to the size of the neighborhood as well as the given orientation.

In this example, we have implemented the operation by applying fixed filters to rotated image data. We have selected this implementation as we have found it to be the solution of choice for several researchers in image processing. As such, the implementation reflects parallelization problems encountered in a realistic situation. It should be noted, however, that the alternative sequential implementations presented in Chapter 7 yield better sequential as well as parallel performance.

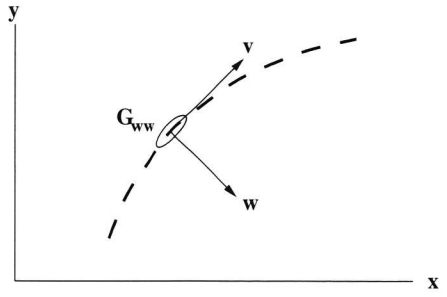


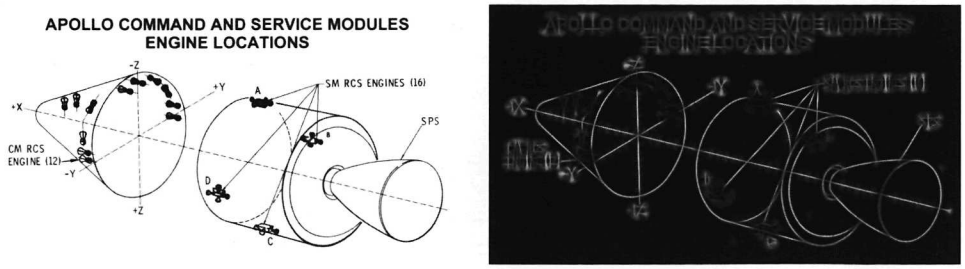
Figure 4.4: *Directional filtering for line detection. Filter* $\mathbf{G_{ww}}$ *is oriented in the line direction; local coordinate system indicated by* $(\mathbf{v}, \mathbf{w})$.

The main body of the sequential implementation is presented in pseudo code in Listing 4.1. The program starts by rotating the original input image for a given orientation θ. In addition, for all (σ_v, σ_w) combinations the filtering is performed by six library operations executed in sequence. First, $f_{ww}^{\sigma_v,\sigma_w,\theta}$ and $b^{\sigma_v,\sigma_w,\theta}$ (or `Filtered1_IM` and `Filtered2_IM`, respectively) are produced by executing two generalized convolutions, each with the appropriate parameters. For cost effectiveness the Gaussian convolutions are performed by applying two 1-dimensional filters in both cases. Next, the result of Equation (4.1) is obtained by executing two binary pixel operations, one having an image, the other having a constant value as argument. Finally, the result image is rotated back to match the orientation of the input image, and the maximum response image is obtained.

Figure 4.5(a) gives a typical example of an image that is used as input to the program. The result obtained after applying the program for a reasonably large parameter subspace of $(\sigma_v, \sigma_w, \theta)$ is shown in Figure 4.5(b). On a state-of-the-art sequential machine the program may take from a few minutes up to several hours to complete, depending on the size of the input image and the extent of the chosen parameter subspace. Consequently, for the directional filtering program parallel execution is highly desired.

```
FOR all orientations θ DO
    Rotated_IM = GeometricOp(Original_IM, "rotate", θ);
    FOR all smoothing scales σ_v DO
        FOR all differentiation scales σ_w DO
            Filtered1_IM = GenConvOp(Rotated_IM, "gauss", σ_w, σ_v, 2, 0);
            Filtered2_IM = GenConvOp(Rotated_IM, "gauss", σ_w, σ_v, 0, 0);
            Detected_IM = BinPixImArgOp(Filtered1_IM, "absdiv", Filtered2_IM);
            Detected_IM = BinPixValArgOp(Detected_IM, "mul", σ_v × σ_w);
            BackRotated_IM = GeometricOp(Detected_IM, "rotate", −θ);
            Contrast_IM = BinPixImArgOp(Contrast_IM, "max", BackRotated_IM);
```

Listing 4.1: *Pseudo code for the directional filtering program.*



(a) (b)

Figure 4.5: *(a) Typical 1000× 554 input image obtained from the Apollo training manual "Apollo Spacecraft & Systems Familiarization" (March 13, 1968). National Aeronautics and Space Administration (NASA), Office of Policy and Plans, NASA History Office. Used by kind permission. (b) Maximum response image obtained after application of the directional filtering program.*

4.5.2 Parallel Execution

As all parallelization issues are shielded from the user, the pseudo code of Listing 4.1 directly constitutes a program that can be executed in parallel as well. Optimization of the efficiency of the program is to be taken care of by the software architecture's scheduling component. For this evaluation, however, we have created two different schedules for the program by hand. In the first schedule *all* library operations are forced to run in a data parallel manner, using all available processors. The second schedule differs from the first in that the last two operations in the innermost loop of the program are run on one node only.

In both schedules the `Original_IM` structure is broadcast to all nodes. This is because the structure is applied in the initial rotation operation, which expects it to have a data access pattern of type 'other' (see Section 3.4). This broadcast needs to be performed only once, as `Original_IM` is not updated in subsequent operations. In addition, in both schedules the first four operations in the innermost loop are executed locally on partial image data structures. The only need for communication is in the exchange of shadow regions in the two Gaussian convolution operations.

In the first schedule the last two operations in the innermost loop are run in parallel as well. This requires the distributed image `Detected_IM` to be available in full at each node, because it has an access pattern of type 'other' in the back-rotation operation. This is achieved by executing a gather-to-all operation, which is logically equivalent to a gather operation followed by a broadcast. Finally, a partial maximum response image `Contrast_IM` is calculated on each node, which requires a final gather operation to be executed just before termination of the program. In the second schedule the last two operations are not executed in parallel. As a result, the intermediate result image `Detected_IM` is gathered to the single node that produces both the back-rotated image, as well as the complete maximum response image.

It is the purpose of the architecture's scheduling component to pick the optimal solution out of multiple competing schedules of this kind. In the following we will show that the APIPM-based performance models are powerful enough to allow the scheduler to make such decisions correctly.

4.5.3 Performance Evaluation

To initialize the APIPM-based performance models we have performed a small set of benchmarking operations. For each instruction used in the directional filtering program not more than two measurements were performed, i.e. for input sizes of 200^2 and 1000^2 elements. Model predictions for each instruction and for each required input size were obtained as indicated in Section 4.4.

The benchmarking operations, as well as the directional filtering program were executed on the 24-node homogeneous DAS-cluster (Distributed ASCI Supercomputer [7]) located at the University of Amsterdam. All nodes in the cluster contain a 200 Mhz Pentium Pro with 64 MByte of EDO-RAM, and are connected by a 1.2 Gbit/sec full-duplex Myrinet SAN network. The nodes run the RedHat Linux 6.2 operating system. At the time of measurements, 4 nodes in the system were unusable. As a consequence, performance results are presented only for up to 20 processors.

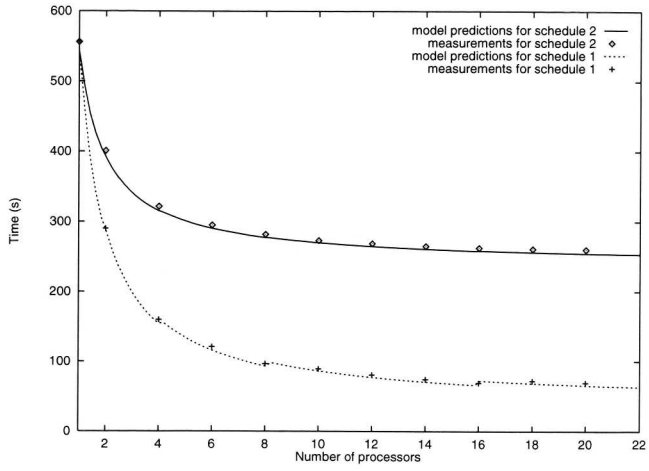
Based on intuition alone a programmer would have great difficulty deciding which of the two schedules described in the previous section should be executed. Clearly, a schedule is preferred if the set of operations unique to that schedule is faster than the set of operations unique to another schedule. Hence, for the directional filtering program the first schedule is preferred if:

$$\begin{aligned} \theta\sigma(P_{rotate}(size/N) + P_{max}(size/N) + P_{bcast}(size)) + P_{gather}(size) \; < \\ \theta\sigma(P_{rotate}(size) + P_{max}(size)) \end{aligned} \tag{4.3}$$

where N denotes the number of nodes, and $\theta\sigma$ the size of the parameter subspace. For the first schedule the large number of broadcasts is expected to have a significant impact on performance. For the second schedule the many rotations of non-partitioned image data is expected to be costly.

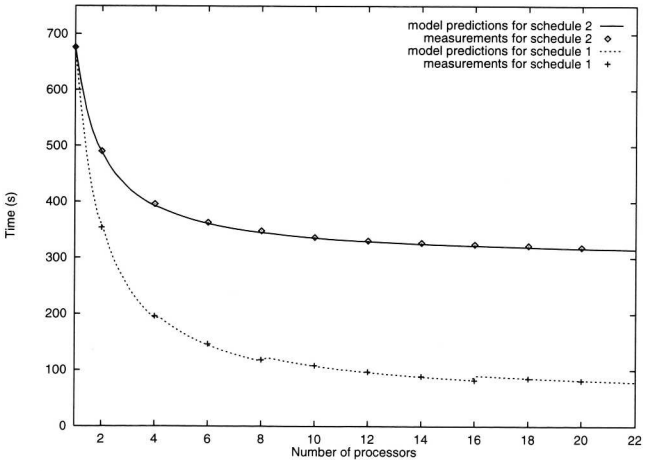
Based on the benchmarking results we are able to decide which schedule is optimal. As shown in Figure 4.6 (depicting the *complete* execution time of both schedules), our models indicate that the first schedule is always preferred - for any number of processors. Clearly, broadcasting a full-sized image structure is not as expensive as performing the image rotation sequentially on one node. The 'hops' in the graph of schedule 1 are explained by the fact that the broadcast operation is implemented using a spanning binomial tree (SBT), which has a cost related to $logN$. Figure 4.7 shows similar predictions for a smaller input image, but for a larger parameter subspace.

To test the accuracy of our performance models we have executed the directional filtering program for both schedules. The resulting mean execution times for each run are included in the graph of Figure 4.6 as well. Error bars are not shown, as the performance of the DAS is quite stable. In most situations measured lower and upper bounds are within 0.5 seconds of the mean execution times. The presented results indicate that the model predictions for both schedules are highly accurate - for any



Schedule 1		
# CPUs	Predicted	Measured
1	543.509	556.696
2	281.690	290.105
4	153.949	159.903
6	115.861	120.952
8	93.247	96.851
10	86.417	89.342
12	77.371	80.914
14	70.910	74.618
16	66.064	69.397
18	68.833	72.411
20	65.818	69.579
Schedule 2		
# CPUs	Predicted	Measured
1	543.387	556.112
2	391.157	401.020
4	315.106	321.753
6	290.104	294.923
8	277.146	281.777
10	269.893	273.113
12	264.709	268.763
14	261.007	265.181
16	258.230	262.711
18	256.396	261.183
20	254.668	260.249

Figure 4.6: *Comparison of model predictions and measurements for the two program schedules. Results for directional filtering of extended Apollo image of size 1098× 1098, and for a parameter subspace including 12 orientations and 4 (σ_v, σ_w) combinations.*



Schedule 1		
# CPUs	Predicted	Measured
1	676.708	676.410
2	350.819	353.654
4	191.762	195.665
6	144.254	146.255
8	116.121	118.161
10	107.507	108.406
12	96.254	97.090
14	88.216	88.965
16	82.187	81.915
18	85.519	85.076
20	81.767	81.173
Schedule 2		
# CPUs	Predicted	Measured
1	676.658	675.980
2	487.080	490.068
4	392.318	396.005
6	361.092	363.335
8	344.964	348.405
10	335.829	337.166
12	329.378	330.883
14	324.769	326.770
16	321.313	323.939
18	318.923	321.811
20	316.773	319.036

Figure 4.7: *Comparison of predictions and measurements for input image of size 707× 707, and for a parameter subspace including 36 orientations and 4 (σ_v, σ_w) combinations.*

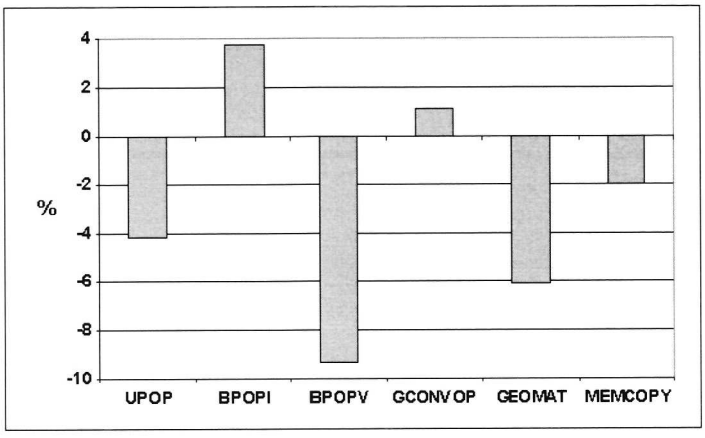


Figure 4.8: *Difference between accumulated predictions and measurements for the six most important APIPM instructions (schedule 1, not including communication).*

number of processors. Even worst case predictions are within 5.5% of the measured values. It is noteworthy, however, that our models are slightly optimistic in all situations. This is explained by the fact that the mean performance values measured in the benchmarking process tend to be somewhat lower than what is actually obtained at application run time. This is because 'outliers' obtained during benchmarking are not included in our database of performance values, while extremely high values may still occur during normal runs of a particular application.

The graph of Figure 4.8 shows that our performance models are capable of providing accurate estimations at the lowest level of APIPM instructions as well. The accumulated estimations on a per-instruction basis are optimistic as well as pessimistic, depending on the applied instruction. The importance of this graph, however, lies in the fact that errors in the estimations for the most significant instructions applied in the application of Listing 4.1 are, in general, not more than 10%. As a consequence, we feel that a sufficiently high level of estimation accuracy is obtained for the models to be applied in our software architecture's optimization process.

Given schedule 1, it can be derived from the models that the impact of communication (especially the repeated broadcast) on overall application performance is huge. Figure 4.9 shows that for 16 nodes the program spends almost half of its time communicating. For 64 nodes 84.1% of the time is lost in all communication steps combined, and 71.1% in broadcasting alone. Although parallel performance is often significantly better for alternative sequential implementations of this particular line detection problem (see Chapter 7), communication costs do play an important role in almost any parallel application. Therefore, it is essential for our performance models to also provide accurate estimations for the SEND and RECV instructions. The next chapter of this thesis is devoted entirely to the modeling of these basic communication operations, and includes a detailed evaluation of our performance estimations as well.

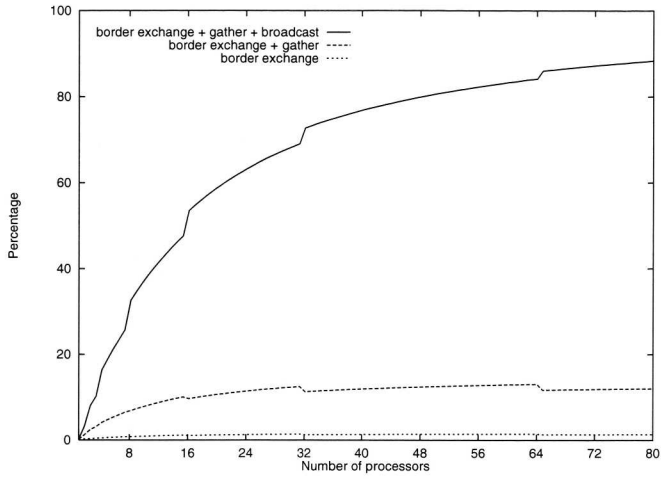


Figure 4.9: *Predicted impact on communication (for schedule 1).*

4.6 Conclusions

In this chapter we have described the performance estimation approach as applied in our software architecture. We have introduced the notion of *semi-empirical modeling*, which is a performance estimation technique based on a combination of relevant *abstraction*, simple *modeling*, and domain-specific *measurement*. We have compared the technique with existing estimation approaches, and have shown semi-empirical modeling to be similar to a combination of two techniques described in the literature: (1) the AFM-based approach of narrow spectrum benchmarking that incorporates very high level system abstractions, and (2) the approach of adaptive sampling that captures system variance by measuring execution times for multiple workloads.

We have indicated that in our semi-empirical modeling approach all abstractions are introduced on the basis of a high level abstract machine specification for parallel image processing (the APIPM), and a related instruction set. Also, we have shown the definition of the abstract machine to reflect the relevant hardware components and behavior common to all projected target platforms for our software architecture (i.e., state-of-the-art homogeneous commodity clusters). Subsequently, we have indicated how to define a simple, linear performance model on the basis of the APIPM-abstractions. Finally, we have shown how domain-specific benchmarking is incorporated for estimation of system variance.

A comparison of model estimations and experimental measurements has indicated that, for a realistic image processing application, the APIPM-based performance models are highly accurate. The models are capable of providing good estimations for full applications, as well as for any constituent subtask. Given these results we are confident in that the core of our software architecture forms a powerful basis for automatic optimization of a wide range of parallel low level image processing applications.

The evaluation of our performance models as presented in this chapter is not complete, as the costs related to interprocess communication hardly have been touched upon. Because communication is such a prominent cost factor in many parallel applications, all modeling aspects related to this issue are deferred to the next chapter. Also, the evaluation has not shown that optimization on the basis of our performance models indeed results in highly efficient parallel applications. All issues related to application optimization and scheduling are discussed extensively in Chapter 6. An assessment of the effectiveness of our software architecture in providing significant performance gains is presented in Chapter 7. Finally, in the appendix to this chapter a more detailed overview is given of the APIPM instruction set, and the related model parameterization.

4.A APIPM Instruction Set Definition

This section presents a detailed discussion of the APIPM instruction set definition. A complete overview of the instructions and their related operands is given in Tables 4A.1 - 4A.3.

Memory References

In the Abstract Parallel Image Processing Machine multiple real-world objects need to be represented. The most prominent objects are images, but templates, matrices, and the likes, are essential as well. In the instruction set we do not introduce a special data representation for each of these objects. Instead, we use *memory references* that contain information about the internal data representation, but lack any information on the semantics of the data referenced to. The semantics are determined by the APIPM instruction the memory reference is passed to as a parameter.

Given the notion of memory references, the operands (arguments) of the instructions fall into one of four categories:

1. *Memory references to single data elements* (`smref`). Operands of this type refer to single data elements stored in main memory. Apart from a pointer to a memory location, it holds information regarding the *scalar type* and *dimensionality* of the data element stored. In a realistic program a single memory reference usually represents a pixel value of a certain scalar type and dimension.

2. *Memory references to aggregated data elements* (`amref`). Operands of this type refer to aggregations (such as arrays) of data elements stored in main memory. Apart from a pointer to the memory location containing the first data element, it also contains information regarding the *size* and *origin* of the domain of the aggregated structure, in combination with the *type* and *dimensionality* of the structure's elements. The size and origin of an n-dimensional aggregated structure are both represented by an n-dimensional vector.
   A memory reference of this type pointing to a data aggregation of size 1 (thus: containing only one element) is considered equal to the single data element

opcode	**operands** memory instructions
CREATE	amref, vector, vector, string, value #1: reference to destination data structure #2: domain size of destination structure #3: domain origin of destination structure #4: scalar type of data elements in destination structure #5: dimensionality of data elements in destination structure
DELETE	amref #1: reference to source data structure
MEMCOPY	amref, amref, vector, vector, value #1: reference to source data structure #2: reference to destination data structure #3: offset from start of source data structure #4: offset from start of destination data structure #5: number of data elements
opcode	**operands** generic image processing instructions
UPOP	amref, amref, string #1: reference to source data structure #2: reference to destination data structure #3: name of internal unary pixel operation
BPOPV	amref, amref, string, smref #1: reference to source data structure #2: reference to destination data structure #3: name of internal binary pixel operation #4: reference to single argument value
BPOPI	amref, amref, string, amref #1: reference to source data structure #2: reference to destination data structure #3: name of internal binary pixel operation #4: reference to argument data structure
REDUCOP	amref, smref, string #1: reference to source data structure #2: reference to single destination value #3: name of internal reduction operation

Table 4A.1: *APIPM instruction set.*

opcode	**operands** generic image processing instructions
NEIGHOP	amref, amref, string, amref #1: reference to source data structure #2: reference to destination data structure #3: name of internal neighborhood operation #4: reference to kernel structure
GCONVOP	amref, amref, string, string, amref #1: reference to source data structure #2: reference to destination data structure #3: name of internal binary pixel operation #4: name of internal reduction operation #5: reference to kernel structure
GEOMAT	amref, amref, amref, smref, string, vector #1: reference to source data structure #2: reference to destination data structure #3: reference to transformation matrix #4: reference to single background value #5: interpolation type #6: translation vector
GEOROI	amref, amref, vector, smref #1: reference to source data structure #2: reference to destination data structure #3: offset from start of source data structure #4: reference to single background value
opcode	**operands** I/O instructions
IMPORT	amref, value, value #1: reference to destination data structure #2: unique external data structure identifier #3: unique external device number
EXPORT	amref, value, value #1: reference to source data structure #2: unique external data structure identifier #3: unique external device number

Table 4A.2: *APIPM instruction set (continued).*

opcode	operands communication instructions
SEND	amref, vector, value, value #1: reference to source data structure #2: offset from start of source data structure #3: number of data elements #4: unique destination ASIPM identifier
RECV	amref, vector, value, value #1: reference to destination data structure #2: offset from start of destination data structure #3: number of data elements #4: unique source ASIPM identifier

legend:

amref	memory reference to aggregated data elements
smref	memory reference to single data element
string	string value (constant)
value	numerical value (scalar)
vector	numerical value (vector)

Table 4A.3: *APIPM instruction set (continued).*

reference described above. References to aggregations of size 1 and references to single data elements can be interchanged at will. In a realistic program a memory reference of this type usually refers to image data.

3. *Numerical (constant) values* (`value` and `vector`). Operands of this type refer to single numbers or vectors of single numbers, and are used to represent sizes, positions, etcetera.

4. *String (constant) values* (`string`). Operands of this type refer to character strings recognized by each sequential image processing unit (SIPU). A string value determines the behavior of an instruction, and is either used as an *operation indicator*, or as a *type indicator*.
   Operation indicators refer to internal operations recognized by the SIPU. As an example, indicators such as "NEGATE" and "SQRT" can be used to represent valid unary pixel operations.
   Type indicators represent additional information required for an operation to be executed. For example, the memory allocation instruction "CREATE" needs an indicator for the specification of the datatype of each element in the structure to be allocated. Also, the geometric transformation instruction "GEOMAT" needs an indicator for the specification of the type of interpolation to be used.

Sequential Instructions

The instruction set contains memory operations to allocate and free memory space ("CREATE" and "DELETE"), and to copy data from a source area to a destination area ("MEMCOPY"). As described above, references to single data elements are equal to aggregated data structures of size 1. This means that a newly created aggregated data structure of size 1 can be used as an argument to an instruction that requires a single data element reference as one of its operands (such as the 'smref' operand in the "REDUCOP" instruction).

Two I/O operations are available in the instruction set to transport data to and from external devices ("IMPORT" and "EXPORT"). Apart from a reference to aggregated data, both instructions need a unique device number and identifier to specify the apparatus itself and the data structure residing on that device.

The generic image processing operations in the instruction set are those we have discussed in Chapter 3. Although we have indicated that many image processing operations can be performed in-place, in all but the reduction operation ("REDUCOP") both a source image reference and a destination image reference are required. This scheme does not introduce additional copying of data because the same reference could be given as an argument twice.

As stated in Section 4.3.2, all image processing operations that have a reference to argument data structures or kernel structures as an operand require *data element homogeneity*, to acknowledge the differences between operations on homogeneous and heterogeneous types. Homogeneity means that the scalar type and the dimensionality of the data elements of both the source structure and the additional structure must be identical. Data element homogeneity is not required for destination image data structures, as the resulting scalar type and dimensionality of the data elements is determined by the type of internal SIPU instruction performed.

Communication Instructions

The instruction set includes two communication operations for the exchange of data between two ASIPMs. Data can be sent to another ASIPM by using the "SEND" operation. Data can be received from another ASIPM by using the "RECV" operation. These point-to-point operations provide *reliable* message transfer. This means that a message sent is always received correctly, and that no additional checks for errors are needed. Collective communication operations (i.e.: communication routines that involve multiple ASIPMs) are not included in the instruction set. This is because a complete set of collective communication routines can be created using the two point-to-point communication operations.

The first operand in the "SEND" operation specifies a *send buffer* in the main memory of the sending ASIPM from which the message data is taken. The starting point in the send buffer and the number of data elements to be sent are specified by the second and third operand. The last operand specifies the unique identifier of the receiving (destination) ASIPM. The scalar type and dimensionality of the data elements sent are specified in the reference to the source data structure.

The "RECV" operation requires a *receive buffer* to store incoming message data. Similar to the "SEND" operation, the offset in the receive buffer and the number of data elements to be received are specified by the second and third operand. Also, the last operand specifies the unique identifier of the sending (source) ASIPM.

All send and receive operations must be *matched*. This means that for each message sent the destination node needs to execute a receive operation with the sending node as source identifier. Furthermore, in both the send call and the receive call the number of data elements, as well as with the scalar type and dimensionality of the elements all must be identical. Also, all messages that are sent over the communication channels are *non-overtaking*. This means that if one ASIPM sends two messages in succession to the same destination, the first message will always be received first.

We assume that the two communication operations are *blocking*. For the "SEND" operation this means that it does not return until the message has been copied into a matching receive buffer (on another ASIPM), or stored away safely in a local temporary buffer. For the "RECV" operation this means that it does not return until the message has been fully copied into its receive buffer. Although not expected under normal circumstances, it is possible for a receive call to complete before its matching send call has completed.

4.B APIPM Model Parameterization

This section presents a detailed discussion of the APIPM model parameterization. A complete overview of the parameterized model instructions and related performance values is given in Tables 4B.1 and 4B.2.

Parameterized Model Instructions

As stated in Section 4.4, a problem with our simple, linear performance model is that most APIPM instructions are not single static entities. This is because the execution of an instruction often depends on the values of its operands. Therefore, a static entity for each possible operand combination must be incorporated in the model. To avoid an explosion of the number of static entities we allow each instruction I_i to be *parameterized* instead. The number of parameters for a model instruction is not necessarily identical to the number of operands of the related APIPM instruction. For example, the background value required in the geometric transformation instructions (Table 4A.2, operand `#4` in either "GEOMAT" or "GEOROI") does not call for additional static model instructions. The execution of these instructions is expected to be independent of the applied background pixel value.

Essentially, any possible source of relevant change in instruction behavior must be captured in a model parameter. Here, the task of choosing relevant model parameters is steered by the actual implementations of each APIPM instruction in our software library. An overview of the parameterized model instructions related to all sequential APIPM instructions is presented in Table 4B.1. The communication instructions have been left out, as parameterization of these instructions (a.o., including the memory layout of a message, see Chapter 5) is outside the scope of this chapter.

instruction	parameterized model instruction
UPOP	I_{UPOP}(opname, idim, stype, ival)
BPOPV	I_{BPOPV}(opname, idim, stype, ival, aval)
BPOPI	I_{BPOPI}(opname, idim, stype, ival, aval)
REDUCOP	$I_{REDUCOP}$(opname, idim, stype, ival)
NEIGHOP	$I_{NEIGHOP}$(opname, idim, kdim, stype, ival, kval)
GCONVOP	$I_{GCONVOP}$(popname, ropname, idim, kdim, stype, ival, kval)
GEOMAT	I_{GEOMAT}(idim, stype, ival, mtype, itype)
GEOROI	I_{GEOROI}(idim, stype, ival)
CREATE	I_{CREATE}(idim, stype)
MEMCOPY	$I_{MEMCOPY}$(idim, stype)
DELETE	I_{DELETE}(idim, stype)
IMPORT	I_{IMPORT}(idim, stype)
EXPORT	I_{EXPORT}(idim, stype)

legend:

aval	value indicator of argument data structure
idim	dimensionality of source data structure
itype	type of interpolation used in geometric operation
ival	value indicator of source data structure
kdim	dimensionality of kernel data structure
kval	value indicator of kernel data structure
mtype	type of matrix used in geometric operation
opname	name of internal operation
popname	name of internal binary pixel operation
ropname	name of internal reduction operation
stype	scalar type of data elements

Table 4B.1: *Parameterized model instructions.*

The parameters for the model instructions presented in Table 4B.1 fall into one of four categories:

- *Type indicators.* Three different type indicators are introduced: 'stype', 'mtype', and 'itype'. The scalar type parameter can have any of the values "byte", "short", "int", "float", and "double". The type of the data structure elements often will have an important impact on the behavior of an instruction. As an example, arithmetic floating point operations (such as sqrt) are often much more expensive than the non-floating point versions.
  The interpolation type indicator is applied in geometric transformation operations, and can have any of the values "nearest" and "linear". The matrix type indicator decides which geometric transformation is performed. Currently, the available valid values are "rotate", "reflect", and "scale".

- *Value indicators.* Three different value indicators are introduced: 'ival', 'aval', and 'kval'. This is because the actual values present in a data structure may have an important impact on the performance of an instruction. For example, when the value '1' is presented to a base-10 logarithm operation '0' is returned. However, if '0' is presented to the operation an error value is returned, and an exception may be raised, possibly causing additional overhead. Currently valid values for the parameters 'ival', 'aval', and 'kval' are:

  - "ALL0" (most elements have the value 0),
  - "ALL1" (most elements have the value 1),
  - "0TO1" (most elements have a value between 0 and 1), and
  - "ANY" (no value indication, used by default).

- *Dimensionality indicators.* Two dimensionality indicators are introduced: 'idim' and 'kdim'. In our software library a choice has been made to provide different implementations for operations on 2-dimensional and 3-dimensional images. Also, multiple fundamental kernel operations have been implemented to allow for optimization of operations that make use of separable kernel data. For this reason the 'idim' parameter can have the value "2D" or "3D". The 'kdim' parameter either can have the value "1D" or "nD". Here we use "nD" to represent non-separable kernel dimensionality.

- *Operation indicators.* Operation indicators refer to the internal operations recognized by the Sequential Image Processing Unit (SIPU). See also Section 4.A.

Parameterized Performance Values

The performance values in set **P** are not single static entities either. This is because the execution time of many instructions is dependent on the size of the workload. For this reason we also have parameterized the performance values related to the model instructions. In Table 4B.2 an overview is given of the parameterized performance values related to the sequential operations in the APIPM instruction set.

parameterized performance values

$P_{I_{UPOP(opname,idim,stype,ival)}}$(ddim, w, h, d)
$P_{I_{BPOPV(opname,idim,stype,ival,aval)}}$(ddim, w, h, d)
$P_{I_{BPOPI(opname,idim,stype,ival,aval)}}$(ddim, w, h, d)
$P_{I_{REDUCOP(opname,idim,stype,ival)}}$(ddim, w, h, d)
$P_{I_{NEIGHOP(opname,idim,kdim,stype,ival,kval)}}$(ddim, w, h, d, kw, kh, kd)
$P_{I_{GCONVOP(popname,ropname,idim,kdim,stype,ival,kval)}}$(ddim, w, h, d, kw, kh, kd)
$P_{I_{GEOMAT(idim,stype,ival,mtype,itype)}}$(ddim, w, h, d)
$P_{I_{GEOROI(idim,stype,ival)}}$(ddim, w, h, d)

$P_{I_{CREATE(idim,stype)}}$(ddim, w, h, d)
$P_{I_{MEMCOPY(idim,stype)}}$(ddim, w, h, d)
$P_{I_{DELETE(idim,stype)}}$(ddim, w, h, d)

$P_{I_{IMPORT(idim,stype)}}$(ddim, w, h, d)
$P_{I_{EXPORT(idim,stype)}}$(ddim, w, h, d)

legend:

ddim	dimensionality of data elements (usually: pixels)
w, h, d	extent of structure's domain in each of 3 dimensions
kw, kh, kd	extent of kernel's domain in each of 3 dimensions

Table 4B.1: *Parameterized performance values.*

The 'ddim' performance value parameter relates to the dimensionality of the data elements (i.e., pixels) of the structure passed to a given instruction. All other performance value parameters relate to the sizes of the data structures passed to an instruction. Although the software architecture's benchmarking component incorporates all, by default the size and dimensionality parameters are taken together to form a single 'total size' parameter. It should be noted that the performance estimations presented in Section 4.5 are all based on benchmarking results obtained after this type of parameter merging.

Chapter 5

A Communication Model for Automatic Decomposition of Regular Domain Problems*

"Mind the gap!"

(warning message broadcast across platforms at London Underground)

One of the most fundamental problems any automatic parallelization and optimization tool is confronted with is to find an optimal domain decomposition for an application at hand. For regular domain problems (such as simple matrix manipulations) this task may seem trivial. However, communication costs in programs executing on commodity clusters often significantly depend on the capabilities and particular behavior of the applied message passing primitives. As a consequence, straightforward domain decompositions may deliver non-optimal performance.

Whereas many software libraries exist that provide efficient message passing implementations [53, 102], MPI seems to have become the de facto standard [104]. Of the large number of functions defined in MPI 1.1, the two blocking point-to-point communication operations (i.e., `MPI_Send()` and `MPI_Recv()`) are most important and most often used (see also Section 2.2.1). To implement optimal parallel applications it is essential to have a thorough understanding of the performance characteristics of these basic communication operations. A good way to make such characteristics explicit is to design a performance model that captures typical point-to-point communication behavior. Because a fundamental MPI design criterion was *portability* across a wide

*This chapter combines our papers published in *Proceedings of the Tenth Euromicro Workshop on Parallel, Distributed and Network-based Processing (PDP 2002)* [142], *IEEE Transactions on Parallel and Distributed Systems* [143] and *Journal of Systems Architecture* [144].

range of computers, such a model must be *applicable* to the same range of machines. Essentially, this implies that a performance model must incorporate a similar level of abstraction as introduced in the MPI standard.

In the literature several point-to-point communication models have been described that match the MPI abstractions up to a certain degree (e.g., the Postal Model [11, 21], LogP [38], and LogGP [1]). Although successful in many situations, these models were not designed for communication according to MPI specifically. Consequently, the models do not incorporate all capabilities of MPI's send and receive operations. As an important example, the effect of memory layout on communication costs is ignored completely. This is unfortunate, as the work of Prieto et al. [124, 125] indicates that a change in the spatial locality of messages exchanged using MPI can have a severe impact on the overall performance of an application. The authors state that "*the bandwidth reduction due to non-unit-stride memory access could be more significant than the reduction due to contention in the network*". Independently, we have come to similar conclusions [139]. Given these results, it is surprising that no model seems to exist that can account for such costs.

As described in Chapter 4, in our software architecture we rely heavily on performance models to perform the task of automatic parallelization of a particular class of regular domain problems: i.e., low level image processing. As the limitations of existing communication models proved to be too severe, we have designed a new model (called *P-3PC*, or the *Parameterized model based on the Three Paths of Communication*), that closely matches the behavior of MPI's standard point-to-point communication operations. P-3PC bears strong resemblance to the aforementioned models, but due to its additional features it provides more accurate estimations in many essential situations.

First, P-3PC acknowledges the difference in the time either the sender or the receiver is occupied in a message transfer, and the complete end-to-end delivery time. Second, P-3PC makes a distinction between communicating data stored either contiguously or noncontiguously in memory. Finally, P-3PC does not assume a strictly linear relationship between the size of a message being transmitted and the communication costs. Although P-3PC is targeted towards the specific needs in our research, it is general enough to be applicable in other research areas as well.

Hence, the primary research issue addressed in this chapter is formulated as follows: How to design a simple and portable communication model that (1) reflects the relevant capabilities of MPI's standard point-to-point communication primitives, and (2) accurately models the communication costs encountered in low level image processing applications executing in data parallel fashion

This chapter is organized as follows. Section 5.1 discusses the requirements for a model to be applied in our software architecture. Also, two popular communication models are evaluated according to these requirements. The new P-3PC model is introduced in Section 5.2. Section 5.3 shows how P-3PC is applied in the evaluation of communication algorithms executed in a realistic image processing application. In Section 5.4 predictions are compared with results obtained on two clusters, each having a different interconnection network, and a different MPI implementation. Concluding remarks are given in Section 5.5.

5.1 Modeling of Message Passing Programs

In our software architecture all parallelization and optimization issues are to be taken care of automatically, hidden from the user. As explained in Chapters 2 and 4, for this task to be performed correctly we rely on domain-specific performance models that are applied in combination with a benchmarking tool and a separate scheduling component. Based on the models and the measured performance values, it is the task of the scheduler to make optimization decisions regarding:

1. the logical processor grid to map data structures onto (i.e., the actual domain decomposition),
2. the routing pattern for the distribution of data,
3. the number of processing units, and
4. the type of data distribution (e.g., broadcast instead of scatter).

In this chapter we focus on the first two optimization tasks in this list. Once the cost characteristics are available of any routing pattern, given any conceivable domain decomposition, the optimal number of processors and the actual type of data distribution can be derived.

In the following we will investigate the requirements for a communication model to be applied in our software architecture. On the basis of these requirements, we will shortly discuss the two most popular models described in the literature.

5.1.1 Model Requirements

In our software library all communication algorithms are implemented using the standard blocking MPI send and receive operations. Because low level image processing operations tend to have a bulk synchronous parallel behavior [103, 162], usage of any of MPI's additional communication modes will hardly result in a performance improvement, and may even be counterproductive (see also [125]). Also, as MPI's standard collective communication operations do not provide all functionality required in our library[†] we have implemented multiple scatter, gather, and broadcast operations in this manner as well.

In such data exchange operations the combined latency of sending or receiving multiple messages in sequence may be overlapping with the end-to-end latency of each single message. As shown in Figure 5.1(a), such latency differences can be significant. This overlap can be made explicit if a performance model incorporates the following properties:

1. The ability to predict the time a processing unit is busy executing either the `MPI_Send()` or the `MPI_Recv()` operation. As the two communicating nodes

[†]The main problem with many of the operations defined in MPI 1.1 is that a possibility to define fluctuating strides in multiple dimensions is lacking. Although this problem is lifted in the MPI-2 definition [105] (with the introduction of the `MPI_Gatherw()` and `MPI_Scatterw()` operations), as of yet MPI-2 implementations are not generally available.

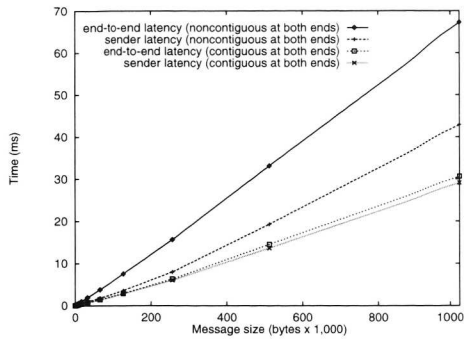
may handle the transfer of data differently (see [16], and also requirement 3 in this section), the communication costs at both ends should be modeled independently.

2. The ability to predict the complete end-to-end latency. Again, the end-to-end latency should be modeled independently from the overhead at either node.

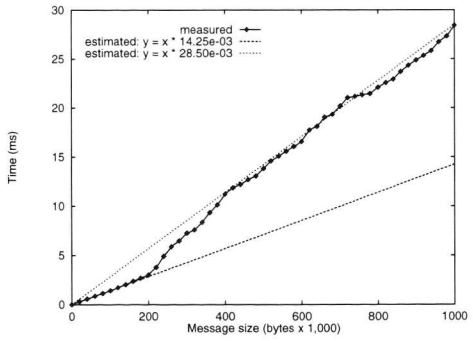
Depending on the type of domain decomposition, it may be necessary to communicate data stored noncontiguously in memory. Using MPI *derived datatypes* it is possible to send such data in a single communication step. As was shown by Prieto et al. [124, 125], knowledge of a message's memory layout is important, as non-unit-stride memory access may have a severe impact on performance due to caching. In addition, the MPI send and receive operations may even handle the transmission of noncontiguous data differently from contiguous blocks. The MPI 1.1 definition [104] states that "*it is up to the implementation to decide whether data should first be packed in a contiguous buffer before being transmitted, or whether it can be collected directly from where it resides*". As shown in Figure 5.1(a) as well, the latency for communicating either contiguous or noncontiguous data may be significantly different indeed. Such differences can be accounted for if a performance model incorporates:

3. The ability to reflect the difference in sending data stored contiguously in memory, and noncontiguous data. Again, the memory layout at the two nodes should be modeled independently.

As a consequence from the fact that the send and receive operations are essentially 'black boxes', it is not safe to assume communication costs to be linearly dependent on message size. As shown in Figure 5.1(b), nonlinearities — caused by caching, buffering, packetization, changes in communication policy, etcetera — may be quite significant. As a final requirement, a model should therefore incorporate:



(a) Latency: sender side versus end-to-end



(b) Sender latency (detailed)

Figure 5.1: *Values obtained on DAS [7] using MPI-LFC [16] (as in Section 5.4).*

4. The ability to provide accurate predictions over a large range of message sizes. For the full range of message sizes a strictly linear increase in communication costs should not be assumed.

In certain application areas it may be important to incorporate *network contention* as well. For our purposes, however, this is not required. In Section 5.5 we will shortly come back to this issue.

5.1.2 Relevant Models in the Literature

In the literature a multitude of message passing models exists. One end of the spectrum consists of models in which communication costs are accounted for by abstracting the interconnection network into a few parameters (e.g., LogP [38], LogGP [1], the Postal Model [11, 21], and the standard linear communication model as described in [50, 79, 114]). Models with a similar level of abstraction are sometimes integrated in a model for computation in order to evaluate architecture and application *scalability* (e.g., the Latency Metric [173]). At the other end of the spectrum are highly parameterized models that are targeted towards a limited set of applications or architectures only (e.g., C^3 [63]).

In our research we must restrict ourselves to models that have an abstraction level comparable to that of MPI. Therefore, models such as the Postal Model, or LogP are seemingly most suitable. As is shown in the following, however, none of these models fully complies with the specific requirements in our research.

The Postal Model

One of the simplest point-to-point communication models is the Postal Model [11, 21], which derives its name from an analogy to the postal service. The model incorporates the notion of *communication latency* through a parameter λ, which represents the inverse ratio of the time it takes a processor to send out a message and the time until the recipient of the message has accepted it. As such, the single parameter captures both the software and the hardware related overhead, such as message preparation, local buffer copying, network propagation delays, and message interpretation. In the

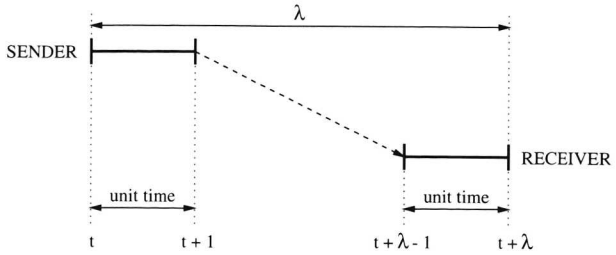


Figure 5.2: *Communication according to the Postal Model.*

model (see Figure 5.2) a *message* refers to an atomic piece of data, which cannot be broken into smaller pieces. The sending of large amounts of data is achieved by sending out several atomic messages in sequence. The time it takes to send or receive a message is defined as one unit of time.

The Postal Model partially adheres to the first two requirements of Section 5.1.1: it acknowledges the difference in the occupation time at each node, and the complete end-to-end latency. However, communication overhead is assumed to be identical at both ends. An assumption of this kind is overly restrictive and is a partial violation of the first two requirements of Section 5.1.1.

The model violates the third requirement as well as it does not allow changes in communication behavior induced by memory layout differences to be made explicit. In addition, the Postal Model uses a single *unit time* for the sending of atomic messages. This assumes a linear growth rate in the time required for sending messages of arbitrary length. This property does resemble the strategy of breaking down large messages into multiple *packets* (as applied by many message passing systems). However, it constitutes a violation of the fourth requirement of Section 5.1.1 as well.

LogP and LogGP

Another communication model that has received considerable attention is the LogP model [38]. The model captures the cost of communicating *small-sized* messages in four parameters:

- L: an upper bound to the *latency* associated with sending a message from one node to another.
- o: the *overhead*, or the amount of time a processor is busy during the transmission or reception of a message.
- g: the *gap*, defined as the minimum time interval between consecutive message transmissions or consecutive message receptions at the same processor.
- P: the number of processor-memory pairs in the machine.

Figure 5.3 shows that LogP bears strong resemblance to the Postal Model. Also, LogP presents a generalization of the Parameterized Communication Model [115], not discussed here.

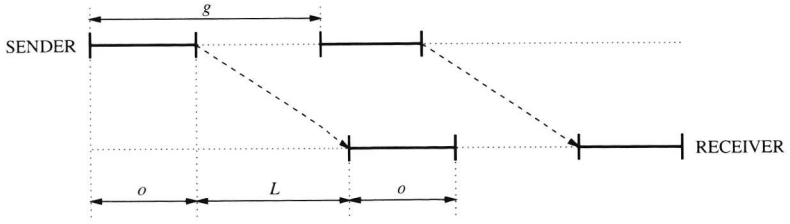


Figure 5.3: *Communication according to LogP.*

In [1] the LogP model is extended with a linear model for long messages. This model, called LogGP, has one additional parameter:

- G: the *gap per byte* (i.e., time per byte) for long messages.

A pictorial view of LogGP is given in Figure 5.4. Clearly, LogGP provides a more accurate description of the communication of long messages than a sequence of LogP communications.

The two models are important because they make explicit the differences in the occupation time at both ends, and the end-to-end delivery time. A possible delay (the gap g) in consecutive transmissions or receipts is accounted for as well. Unfortunately, the two models suffer from the same problems as the Postal Model. First, in both models communication overhead is assumed to be identical at both ends. Second, memory layout differences are not incorporated. Finally, the models assume a strictly linear growth rate in the time required for sending messages of arbitrary length. As a consequence, we conclude that the two models do not comply with the specific needs in our research either.

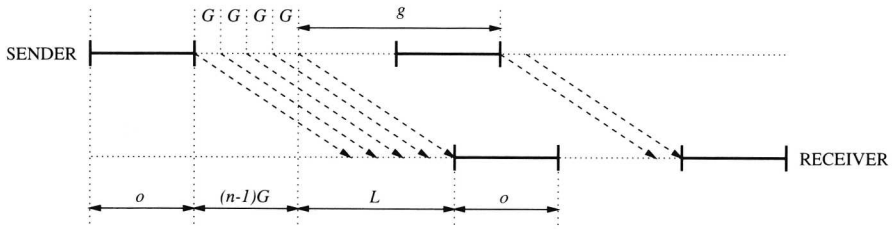


Figure 5.4: *Communication according to LogGP.*

5.2 The P-3PC Model

As no model exists that meets all requirements of Section 5.1.1, we introduce a new communication model. The model, which we refer to as *P-3PC*, or the *Parameterized model based on the Three Paths of Communication*, will be discussed in two parts. First we introduce a simplified version of the complete model (called *3PC*), that complies only with the first two requirements of Section 5.1.1. Subsequently, the 3PC model is extended to incorporate the remaining two requirements.

5.2.1 Part I: 3PC

Given the first two requirements of Section 5.1.1, we introduce the notion of the *three paths of communication*, and assume that the cost of message transmission can be captured in three independent values:

- T_{send}: the cost related to the communication path at the sender (i.e., the time required for executing the `MPI_Send()` operation).

- T_{recv}: the cost related to the communication path at the receiver (i.e., the time required for executing the `MPI_Recv()` operation).

- T_{full}: the cost related to the full communication path (i.e., the time from the moment the sender initiates a transmission until the receiver has safely stored all data and is ready to continue).

For each path we assume that the communication costs can be represented by two parameters. The transmission of any message is expected to involve a constant amount of time, identical to the cost of sending a 0-sized message. This cost is captured by the mutually independent parameters t_{cs}, t_{cr}, and t_{cf} (for the sender, receiver, and full path respectively). At the sender side this value may represent what is often referred to as the *message startup time*, but we prefer not to use this terminology to avoid unnecessary overspecification. Also, for each transmitted byte we assume an 'additional time', which is captured by the mutually independent parameters t_{as}, t_{ar}, and t_{af} respectively. The three communication times (see also Figure 5.5) involved in the transmission of a message containing n bytes are then given by:

$$\begin{aligned} T_{send}(n) &= t_{cs} + n \cdot t_{as}, \\ T_{recv}(n) &= t_{cr} + n \cdot t_{ar}, \\ T_{full}(n) &= t_{cf} + n \cdot t_{af}. \end{aligned}$$

Thus, 3PC simply constitutes a combination of three traditional linear models as also applied in [50, 79, 114]. Note that the manner in which accurate values for the model parameters can be obtained is independent of the actual MPI implementation or the type of communication hardware used. A detailed description of our method of measurement is given in Section 5.4.

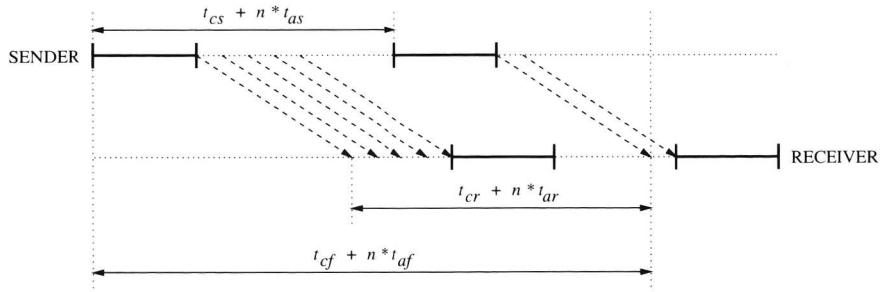


Figure 5.5: *Communication according to 3PC.*

5.2.2 3PC versus LogGP

The LogGP model of Figure 5.4 constitutes a superset of all conventional models of Section 5.1.2. In other words, it is possible to express models such as the Postal Model, or LogP, in terms of the LogGP parameters. For this reason it is relevant to indicate that 3PC preserves the important qualities of LogGP under the following assumptions:

$$\begin{aligned} t_{cs} &= t_{cr} = g, \\ t_{cf} &= 2o + L, \\ t_{as} &= t_{ar} = t_{af} = G. \end{aligned}$$

Because in state-of-the-art communication processors LogGP's o parameter is either negligible [22] or comparable to g (even for relatively small messages, see [92]), 3PC is even *identical* to LogGP under the given assumptions. Compared to LogGP we feel that 3PC is easier to understand, as for each communication path similar parameters are defined. Given the fact that the costs for the three paths of communication are made independent (which is not the case in any of the other models), we conclude that 3PC is expected to be at least as powerful as the LogGP model. Note, however, that we do not claim that 3PC is necessarily a better alternative to LogGP for detailed study of communication behavior. It is introduced only for it to serve as a basis for the P-3PC model.

5.2.3 Part II: P-3PC

To incorporate the last two requirements of Section 5.1.1, the 3PC model is 'parameterized' with a cost indicator M, representing the memory layout at the two communicating nodes. Also, it is assumed that each 'additional time' parameter is a function of n, instead of a constant value for all message sizes. In this extended model (called *P-3PC*), the three communication times involved in a message transfer are given by:

$$\begin{aligned} T_{send,M}(\mathrm{n}) &= t_{cs} + t_{as,M}(\mathrm{n}), \\ T_{recv,M}(\mathrm{n}) &= t_{cr} + t_{ar,M}(\mathrm{n}), \\ T_{full,M}(\mathrm{n}) &= t_{cf} + t_{af,M}(\mathrm{n}), \end{aligned}$$

where $\mathtt{M} \in \{cc,\ cn,\ nc,\ nn\}$. These layout descriptors indicate the four memory layout combinations at the sender and the receiver combined. For example, *cn* means that a contiguous block of data is transmitted by the sender, which is accepted as a noncontiguous block by the receiver.

As no a priori assumptions can be made about the shape of the 'additional time' functions, a set of benchmarking operations must be performed for several different message sizes. As also indicated in Chapter 4, one possibility is to arbitrarily choose a set of relevant message sizes, but an adaptive benchmarking technique could be used as well to actively search for nonlinearities in the communication costs. In any case, based on the benchmarking results (and in accordance with the fourth requirement of Section 5.1.1), each 'additional time' function is assumed to be piecewise linear between each pair of measured communication cost values.

5.3 Application of the P-3PC Model

This section shows how the P-3PC model is applied to evaluate the communication costs involved in one of the most essential applications in image processing: i.e., evaluation of the differential structure of images. Examples are edge detection (based on first and second order derivatives) and invariants (based on i-th order derivatives). Applications of this kind are good examples of regular domain problems as referred to in the work of Prieto et al. [124, 125].

As is well-known, a derivative is best computed using convolution with a separable Gaussian kernel (i.e., n 1-D kernels, each applied in one of the image's n dimensions). The size of the convolution kernel depends on the smoothing scale σ and the order of the derivative. In this example (and in the measurements discussed in the next section) we restrict ourselves to first and second order derivatives (five in total) in the x- and y-direction of 2-D image data, and $\sigma \in \{1, 3, 5\}$. Here, for $\sigma = 1$, the sizes of the 1-D kernels for the i-th order derivative (with $i \in \{0, 1, 2\}$) in any direction are 7, 9, and 9 pixels respectively. For $\sigma = 3$ the kernel sizes are 15, 23, and 25 pixels, and for $\sigma = 5$ these are 23, 37, and 39 pixels respectively. For readers unfamiliar with image processing it is sufficient to know that these kernel sizes partially determine the amount of data exchanged among neighbors in a logical CPU grid — as is explained in more detail below.

When running such application in parallel, three different communication algorithms are to be executed. First, the input image is to be spread throughout the parallel system in a scatter operation. Second, to calculate partial derivative images, pixels in the *border regions* of each partial input image are to be exchanged among neighboring nodes in the logical CPU grid. Finally, after having performed all relevant (application dependent) sequential operations, resulting image data is to be gathered at a single node, for on-screen display or storage.

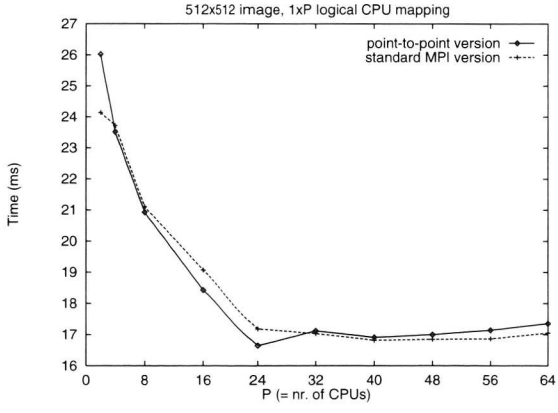
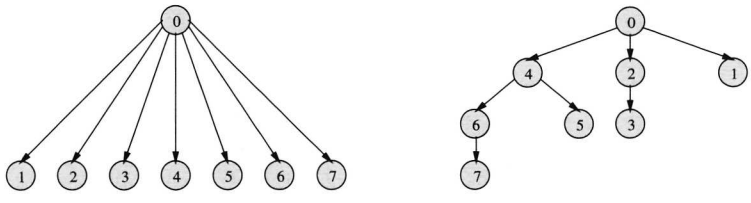


Figure 5.6: *Comparison of MPI_Scatterv() and OFT scatter implemented using MPI_Send() and MPI_Recv() calls (measured on DAS using MPI-LFC).*

As indicated in Section 5.1.1, in addition to the collective operations available in MPI we have implemented multiple scatter and gather operations ourselves, using standard blocking point-to-point operations. As shown in Figure 5.6, our implementations — which, in contrast to the MPI versions, allow definition of fluctuating strides in multiple dimensions — can often compete with available MPI implementations. This indicates that many MPI distributions are not optimized for a particular machine, a problem also discussed in [125, 163]. Of course, in cases where the MPI implementations are faster (and match our specific needs), we apply these versions and use the P-3PC estimations for our fastest implementation as an upper bound. In the following, the modeling of such operations is restricted to two different implementations, one based on a one-level flat tree (OFT), and the other based on a spanning binomial tree (SBT) (see Figure 5.7).

In case of the OFT scatter operation the root sends out data to all other nodes in sequence. If a $1 \times P$ logical CPU grid is assumed (where P is the number of nodes), for each node the data sent out by the root is stored contiguously in memory; for all other grids all data blocks sent out are noncontiguous. In addition, for all possible grids all data is accepted as a contiguous block at each receiving node. As each node in the OFT has to wait for all lower-numbered nodes to be serviced by the root before it will receive data itself, the communication costs are highest at either the root or at the leaf node that is last serviced (depending on the benchmarking results). A worst case P-3PC estimation of this operation is shown in the `timeOFTscatter()` operation in Listing 5.1. An estimation of the related OFT gather operation is simply obtained by setting nc to cn, and changing all occurrences of T_{send} to T_{recv}.

P-3PC estimation of the spanning binomial tree scatter operation is slightly more complicated. In such operation the root node sends out data to $logP$ other nodes. Also, each non-leaf node forwards all received data it is not responsible for. If X is the number of nodes defined in the x-direction of the logical CPU grid, the number of messages involving contiguous data blocks sent out by the root is $logP - logX$; the remaining messages sent out are all noncontiguous. In general, the communication costs will be highest at either the root node, or the node that is $logP$ full communication paths away from the root. The `timeSBTscatter()` operation in Listing 5.1 shows the worst case P-3PC estimation of this operation. An estimation of the related SBT gather operation is obtained as before.



(a) One-level Flat Tree (b) Spanning Binomial Tree

Figure 5.7: *Example communication trees for data scattering.*

```
double timeOFTscatter() {
    M ← (X .eq. 1) ? cc : nc                    // X = nr. of nodes in x-direction of logical CPU grid
    time1 ← (P − 1) · T_send,M(imw · imh/P)      // P = total nr. of nodes
    time2 ← (P − 2) · T_send,M(imw · imh/P) + T_full,M(imw · imh/P)   // imw = image width
    return max(time1, time2)                     // imh = image height
}

double timeSBTscatter() {
    time1 ← 0.0
    time2 ← 0.0
    for (i=1; i.leq.logP - logX; i++)
        time1 ← time1 + T_send,cc(imw · imh/(2 · i))
        time2 ← time2 + T_full,cc(imw · imh/(2 · i))
    }
    for (i=logP-logX+1; i.leq.logP; i++)
        time1 ← time1 + T_send,nc(imw · imh/(2 · i))
        time2 ← time2 + T_full,nc(imw · imh/(2 · i))
    }
    return max(time1, time2)
}

double timeBorderExchange() {                    // bw = border width
    return (2 · T_full,nn(bw · imh) + 2 · T_full,cc((imw + 2 · bw) · bh))   // bh = border height
}
```

Listing 5.1: *P-3PC estimation of OFT & SBT scatter, and border exchange.*

A well-known method to implement Gaussian convolution is to extend the domain of the image structure with a *scratch border* that, on each side of the image in dimension n, has a size of about half the 1-D kernel applied in that dimension. When executed in parallel, neighboring nodes in the logical CPU grid need to exchange pixel values to correctly fill the borders of all extended partial images. In our library, the exchange of border data is executed in four communication step. First, each node sends a subset of its local partial image to the neighboring node on its right side in the logical CPU grid (if such neighbor exists). When a node has accepted this block of data (i.e., after a full communication path period), it subsequently transmits a subset of its local partial image to its left neighbor. As shown in Figure 5.8 these steps in the border exchange algorithm always involve noncontiguous blocks of data. Similarly, in the next two steps border data is exchanged in upward and downward direction, in both cases involving contiguous blocks only. Thus, the `timeBorderExchange()` operation in Listing 5.1 gives a worst case P-3PC estimation for this routine.

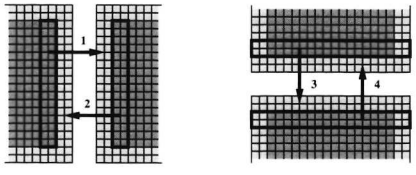


Figure 5.8: *Border exchange (right-left and down-up).*

5.4 Measurements and Validation

To validate the P-3PC model we have performed a representative set of benchmarking operations. For each communication path and memory layout combination measurements were performed using 4 different message sizes, arbitrarily set at 1K, 50K, 100K and 500K (all 4-byte values). Benchmarking was performed for 0-sized messages as well. Note that these values are *not* chosen to best match the communication characteristics for one particular parallel computer. These sizes are representative for messages transmitted in many image processing applications, and are set identically for all machines. Also note that the sizes applied by the architecture's benchmarking tool can be user-defined as well; the sizes given here are used by default.

Clearly, there is a trade-off between the number of benchmarking operations to be performed and the obtainable estimation accuracy. Still, the predefined set of only 4 message sizes is generally sufficient to obtain highly accurate performance estimations for the much larger range of message sizes encountered in a real application. In this respect it is important to note that, in the measurements presented in the remainder of this chapter, actual message sizes range from 192 bytes up to 8 MB.

```
double timePath(int pathType, int bufsize, int sendLayout, int recvLayout, int nrRounds) {
    if (sendLayout .eq. NONCONTIGUOUS)                    // definition of 'sendType'
        MPI_Type_vector(100, bufsize/100, 2*bufsize/100, MPI_FLOAT, &sendType);
    else
        MPI_Type_vector(1, bufsize, bufsize, MPI_FLOAT, &sendType);
                                                          // definition of 'recvType' is similar
    for (i=1:nrRounds) {
        if (myCPU() .eq. 0) {
            if (pathType .eq. SEND) {                     // measure send path
                time1 ← MPI_Wtime();
                MPI_Send(buf, 1, sendType, 1, ...);
                time2 ← MPI_Wtime();
                total ← total + time2-time1;
            } else if (pathType .eq. RECV) {              // measure receive path
                time1 ← MPI_Wtime();
                MPI_Recv(buf, 1, recvType, 1, ...);
                time2 ← MPI_Wtime();
                total ← total + time2-time1;
            } else if (pathType .eq. FULL) {              // measure full path
                time1 ← MPI_Wtime();
                MPI_Send(buf, 1, sendType, 1, ...);
                MPI_Recv(buf, 0, recvType, 1, ...);
                time2 ← MPI_Wtime();
                total ← total + ((bufsize .eq. 0) ? (time2-time1)/2 : (time2-time1)-2·t_cf);
            }
        } else if (myCPU() .eq. 1)
            // matching send and recv calls at node 1 are not shown
    }
    return (total/nrRounds);
}
```

Listing 5.2: *Pseudo code for benchmarking all path-layout combinations. The constant time values t_{cs}, t_{cr}, and t_{cf} are obtained if* `bufsize` *equals zero.*

To give full insight in the benchmarking process, Listing 5.2 gives a simplified overview in pseudo code. To measure communication for noncontiguous data, a fixed number of 100 memory blocks (a conservative estimate of the number of blocks possibly used in a real application, and again a default setting) is combined in a single derived datatype definition. For contiguous data only one block is used in such definition. Measurements for the send and receive paths are obtained by letting one node continuously send data to another node. Full communication path measurements are obtained by subsequently sending out a message of size '`bufsize`', and receiving a 0-sized message. As these operations are similar to those applied by many others in the literature we leave all further interpretation to the reader.

5.4.1 Distributed ASCI Supercomputer (DAS)

The first set of measurements was performed on the 128-node homogeneous DAS-cluster [7] located at the Vrije Universiteit in Amsterdam. All measurements were performed using MPI-LFC [16], an implementation which is partially optimized for the DAS. The 200 Mhz Pentium Pro nodes (with 128 MByte of EDO-RAM) are connected by a 1.2 Gbit/sec full-duplex Myrinet network, and run RedHat Linux 6.2.

The performance values obtained for this machine are presented in Figure 5.9. The values indicate that transmitting noncontiguous data indeed has a significant impact on performance. In this case, the additional overhead is due to the fact that MPI-LFC uses a contiguous send-buffer for noncontiguous data. To preserve the elegance of the benchmarking code, we have measured multiple 'constant time' values for each communication path ($m = 0$). These additional values do not affect the estimations presented in this section in any way.

In the following we show the results as obtained for the example application of Section 5.3. For each of the communication algorithms we have been careful to keep

	m=0	m=1K	m=50K	m=100K	m=500K
$T_{send,cc}(m)$	5.98	61.72	4355.45	10246.77	58596.98
$T_{send,cn}(m)$	8.04	60.74	4363.35	9853.95	57141.29
$T_{send,nc}(m)$	7.93	248.88	5722.00	15142.74	90478.81
$T_{send,nn}(m)$	8.29	133.88	5582.23	14137.45	87870.27
$T_{recv,cc}(m)$	14.86	58.08	5754.93	12037.78	60062.70
$T_{recv,cn}(m)$	14.89	127.30	9527.59	19467.08	98016.47
$T_{recv,nc}(m)$	14.43	46.56	5517.28	12364.45	61446.05
$T_{recv,nn}(m)$	14.82	125.05	9340.63	19685.86	98275.11
$T_{full,cc}(m)$	23.61	131.39	4506.32	11007.89	61277.46
$T_{full,cn}(m)$	25.54	214.10	8665.39	19195.53	97219.23
$T_{full,nc}(m)$	27.05	206.94	6696.30	18015.91	95546.60
$T_{full,nn}(m)$	24.47	287.89	11746.29	25652.54	132399.20

Figure 5.9: *Benchmarking results obtained on DAS (in μs).*

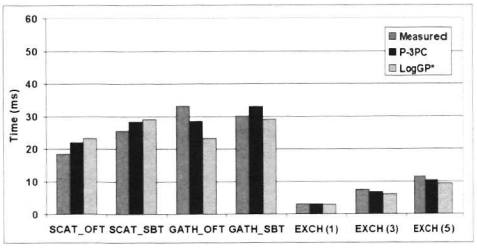
the intrusiveness of the measurements to a minimum. All P-3PC estimations are obtained as in Listing 5.1. Also, in all situations we compare our results with those obtained with LogGP. To avoid using a particularly bad value for the 'G' parameter, we assume a piece-wise linear dependence on message size in the LogGP model as well. In addition, to be able to use the measured values of Figure 5.9, we have reduced the P-3PC model into LogGP in the following manner: $g = t_{cs}$, $L = t_{cf}$, and $G = t_{af,cc}$. As indicated in Section 5.2.2, this reduction makes P-3PC identical to LogGP. Still, to overcome any problem the reader may have with this interpretation of the model, in the remainder we will refer to it as LogGP*.

In Figure 5.10(a) results are presented for a 512^2 floating point image, which is mapped onto a 1×16 logical CPU grid. The graph shows results for the two available implementations of the scatter and gather routines, as well as for the border exchange (for all $\sigma \in \{1,3,5\}$). For such data decomposition all messages involve contiguous blocks only. This is even the case for the border exchange, as no node has a neighbor to its left or right. The graph shows that P-3PC and LogGP* are both quite accurate for this type of data decomposition. As was to be expected, the estimations obtained from the two models are comparable, although P-3PC seems to do marginally better. Apparently, introduction of the three communication paths indeed produces a slightly more accurate model. Here, the differences are marginal, however, and provide no justification for P-3PC's added complexity.

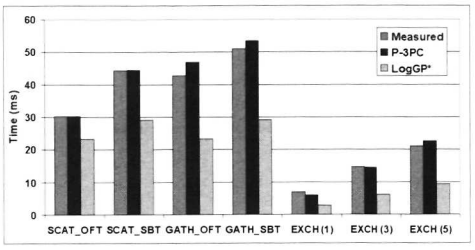
As can be seen in Figure 5.10(b), for a 16×1 data decomposition P-3PC outperforms LogGP* by far. This is because for such decomposition all messages involve noncontiguous data at the sender side. Figure 5.10(c) and Figure 5.10(d) show similar results for 8×1 and 32×1 decompositions. A comparison for larger image data structures is shown in Figure 5.10(e) and Figure 5.10(f). Although most P-3PC estimations are highly accurate, deviations from actual measurements are usually due to small inaccuracies in the performance values obtained by benchmarking. Sometimes, algorithm performance is also slightly degraded by contention in the network — an effect not accounted for by P-3PC. However, the impact of memory layout on performance is always more significant than that of contention. Note that this matches the results of [124, 125].

Figure 5.10(g) and Figure 5.10(h) show that the P-3PC model indeed allows the scheduler of Section 5.1 to make correct optimization decisions. According to the LogGP* model, scattering or gathering a 256^2 floating point image is about as expensive for each communication tree and data decomposition. In practice this is not true, however, and P-3PC gives much more accurate estimations at all times.

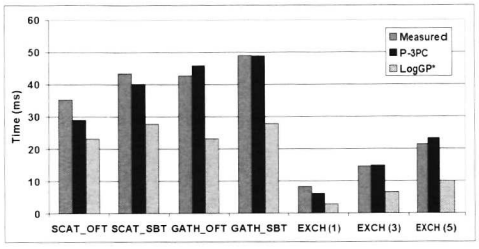
Figure 5.11 gives results for the communication algorithms applied to all possible decompositions involving 16 nodes. Again, P-3PC outperforms LogGP* in almost all situations. It is interesting to see in Figure 5.11(a-d) that, while for all but the 1×16 decomposition P-3PC is somewhat pessimistic, the estimations get better for decompositions that are 'closer' to 16×1. This is explained by the fact that in the benchmarking phase noncontiguous communication is measured using blocks that have quite a significant distance from one another in memory. Thus, caching can become a significant factor, which is indeed expected to be most prominent in a 16×1 decomposition (again, see also [124, 125]).



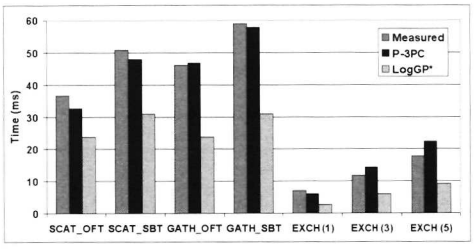
(a) 512^2 image on 1×16 CPU grid



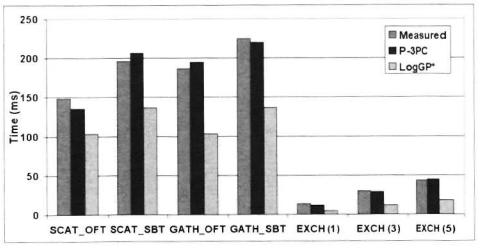
(b) 512^2 image on 16×1 CPU grid



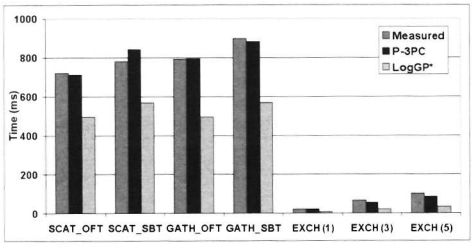
(c) 512^2 image on 8×1 CPU grid



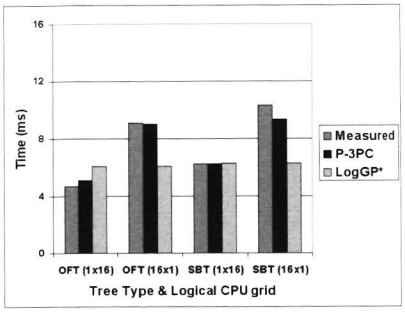
(d) 512^2 image on 32×1 CPU grid



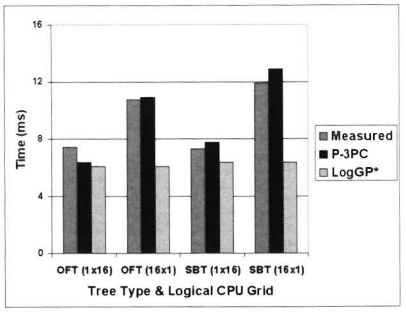
(e) 1024^2 image on 16×1 CPU grid



(f) 2048^2 image on 16×1 CPU grid

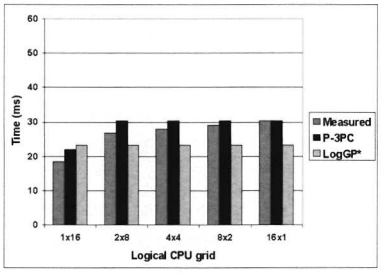


(g) Scatter (256^2 image)

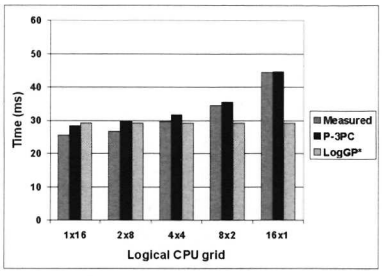


(h) Gather (256^2 image)

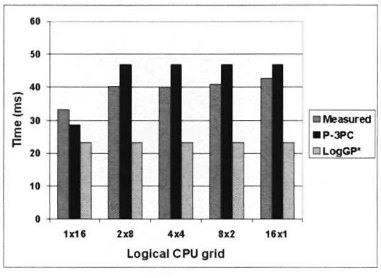
Figure 5.10: *Measurements (DAS) versus P-3PC & LogGP* estimations (1).*



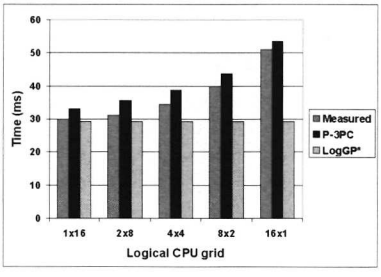
(a) OFT Scatter (512^2 image)



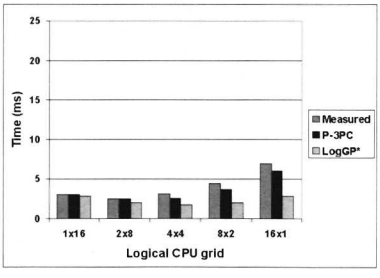
(b) SBT Scatter (512^2 image)



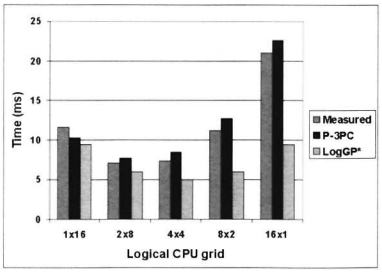
(c) OFT Gather (512^2 image)



(d) SBT Gather (512^2 image)



(e) Exchange (512^2 image, σ=1)



(f) Exchange (512^2 image, σ=5)

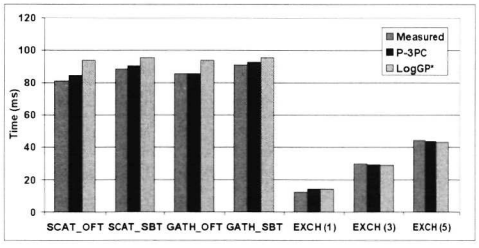
Figure 5.11: *Measurements (DAS) versus P-3PC & LogGP* estimations (2).*

Figure 5.11(e) and Figure 5.11(e) and (f) show that P-3PC gives accurate estimates for the border exchange algorithm for all data decompositions as well. Whereas LogGP* indicates that a 4×4 decomposition is always optimal (which is explained by the fact that the amount of border data is smallest when each partial image is square), P-3PC correctly prefers the 2×8 decomposition. Because the exchange of border data may be performed hundreds of times in a realistic application (for example, see [55] for such application that even applies values of $\sigma > 5$), these results are important indeed. For additional results obtained on the DAS (also including sequential computation) we refer to [147].

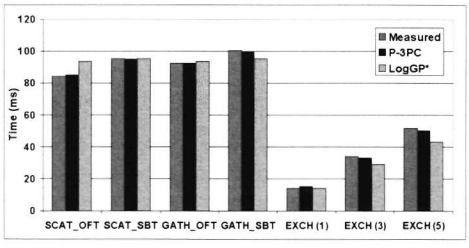
5.4.2 Beowulf at SARA

The second set of tests was performed on the 40-node Beowulf-cluster located at SARA, Amsterdam. On this machine, measurements and benchmarking were performed using MPICH-1.2.0 [61]. The 700 Mhz AMD Athlon nodes (with 256 MByte of RAM) are connected by a 100 Mbit/sec switched Ethernet network, and run Debian Linux 2.2.17.

Because the cluster is heavily used for other research projects as well, we have been able to use only 8 nodes at a time. Figure 5.12 presents results for all algorithms, using a 512^2 floating point image which is mapped onto a 1×8 grid as well as a 8×1 grid. The graphs show that the two models are both quite good in all cases, but P-3PC again provides more accurate estimations. It is clear that the MPICH implementation is much better than the MPI-LFC implementation used on the DAS. Any additional overhead due to non-unit-stride memory access is not caused by buffer copying, but can be attributed to caching alone. Although less significant on the cluster at SARA, this is exactly the effect Prieto et al. have shown to be important on other parallel machines [124, 125].



(a) 512^2 image on 1×8 CPU grid



(b) 512^2 image on 8×1 CPU grid

Figure 5.12: *Measurements (Beowulf at SARA) versus P-3PC and LogGP* estimations.*

5.5 Conclusions

In this chapter we have presented the new P-3PC model for predicting the execution time of communication algorithms implemented using MPI's standard point-to-point operations. P-3PC incorporates the notion of the 'three paths of communication', and accounts for differences in performance at the sender, the receiver, and the full communication path. In addition, P-3PC models the impact of memory layout on communication costs, and accounts for costs that are not linearly dependent on message size. Compared to similar models, P-3PC has the potential for higher predictive accuracy due to its close match with the capabilities and possible behavior of MPI's point-to-point operations.

P-3PC's predictive power is essential to perform the important task of automatic and optimal decomposition of regular domain problems. Although designed for this specific task, we expect the model to be relevant in other research areas as well. It is important to note, however, that P-3PC suffers from the same problem as other models that abstract from the actual network topology (see also [38]). The model can not discriminate between algorithms that cause severe network contention, and those that do not. In our research this is not a problem, as we only apply communication patterns that are expected to perform well on most network topologies used today. Still, because P-3PC is similar to the LogGP model, it can easily be extended to account for contention, in the same manner as described in [3].

It should also be noted that we do not claim the P-3PC model to give a precise characterization of all types of memory access. Any cost factors other than those related to contiguous and noncontiguous memory access are implicit (such as specific cache behavior, differences between programmed I/O and DMA transfer, etcetera), but are still captured due to the semi-empirical modeling approach described in Section 4.2. In this respect, an extension to the P-3PC model that would give a more detailed characterization of non-unit-stride memory access, would be to incorporate a *stride parameter* that captures the actual distances between contiguous blocks transmitted in a single communication step. We have not included such parameter as the results obtained with the current model were shown to be sufficiently accurate.

As the P-3PC model stresses the importance of benchmarking to obtain accurate values for the model parameters, one may argue that the *predictive power* of the model is limited. However, the model does not specifically *enforce* a large number of measurements to be performed. As for models that incorporate a similar level of abstraction, a set of three or four measurements for each communication path may already be sufficient to obtain accurate predictions. The P-3PC model merely acknowledges that nonlinearities in communication costs may be significant (as shown in Section 5.1.1) and should be accounted for.

We are aware of the fact that an evaluation of P-3PC is never complete. However, the evaluation as presented in this chapter — incorporating two fundamentally different interconnection networks, and two different MPI implementations — has shown the model to be highly accurate in estimating the communication costs related to any type of domain decomposition used in a realistic image processing application. As such, we have shown P-3PC to be useful as a basis for automatic and optimal decomposition within the extensive application area of regular domain problems. Also, because P-3PC is capable of modeling behavior that was shown to be problematic in [124, 125], we expect the model to be applicable to the very same machines and MPI implementations as well.

Chapter 6

A Finite State Machine for Global Optimization of Application Performance*

"Speed is good only when wisdom leads the way."

James Poe (1921-1980)

In the previous chapters we have shown how to implement parallel versions of many common image processing operations in a sustainable manner. Also, we have shown how to accurately model the run time performance of such operations. To obtain high performance for complete *applications*, however, it is not sufficient to consider parallelization and optimization of the operations in isolation. This is because parallel code consisting of a sequence of optimized parallel routines often contains many redundant communication steps. Also, in many situations it is possible to further reduce communication overhead by combining multiple messages in a single transfer.

Automatic optimization of communication overhead is not easy. First, this is because the applied optimization strategy must be able to determine which communication steps are essential, and which can be safely combined or removed. In addition, the approach must guarantee that the resulting parallel code is:

- *efficient*, preferably comparable to an optimal hand-coded implementation,
- *legal*, in the sense that the program is deterministic (i.e., always produces the same result) and can never end in deadlock, and
- *correct*, such that it produces output identical to that of the original program.

*This chapter is an extended version of our paper published in *the 17th International Parallel & Distributed Processing Symposium (IPDPS 2003)* [145].

In this chapter we propose a new, and surprisingly simple strategy for global performance optimization that adheres to the stated list of requirements. In the approach, a *fully sequential program* is parallelized automatically by inserting communication operations whenever necessary. The approach, which is referred to as *lazy parallelization*, is based on a simple *finite state machine (fsm)* specification. One of two essential fsm ingredients is a set of states, each corresponding to a valid internal representation of a distributed data structure at run time. The other essential ingredient is a set of state transition functions, each of which defines how a valid internal data structure representation is transformed into another valid representation.

Although it is shown that lazy parallelization works well in many situations, the approach does not guarantee to always produce the *fastest possible* version of a program. First, this is because the approach always applies the fastest communication step whenever message transfer is mandatory. This is a form of *local* performance optimization, however, as it may be better to insert a *combined* message transfer to avoid additional communication at a later stage. Also, the approach does not incorporate knowledge obtained from our APIPM-based performance models (see Chapter 4).

To overcome these problems, this chapter also proposes an extended technique, which requires an *application state transition graph (ASTG)* to be generated for the program under consideration. An ASTG incorporates all optimization decisions that can possibly be made at run time. Each decision is annotated with a cost estimation, such that the fastest program is represented by the 'cheapest' branch in the graph. A drawback of this approach, however, is that it is often costly to obtain the cheapest branch. This is because the ASTG is generally large, even for applications of moderate size. Therefore we also define additional heuristics for search space reduction.

Hence, the primary research issue addressed in this chapter is formulated as follows: How to automatically convert a legal sequential image processing application into a legal, correct, and efficient (preferably even time-optimal) parallel version of the same program? As this issue is the central, most essential problem our software architecture for user transparent parallel image processing is confronted with, the proposed solution incorporates all results obtained in Chapters 3, 4, and 5.

This chapter is organized as follows. Section 6.1 describes the optimization problem. Section 6.2 introduces the finite state machine (fsm) definition. The fsm-based optimization strategy of lazy parallelization is described in Section 6.3. Section 6.4 presents a short description of the ASTG, and some heuristics for search space reduction. In Section 6.5 related work is discussed. Conclusions are given in Section 6.6.

6.1 The Performance Optimization Problem

In Chapter 3 we have defined a default parallelization strategy for each library routine. For operations executed *in isolation*, this default strategy is optimal. This is because communication overhead is minimized, while — for the given parallelization granularity — the available parallelism is fully exploited. When several optimized parallel routines are executed in sequence, however, communication overhead is generally far from optimal. This section explains the problem in more detail.

6.1.1 Abstract Function Specifications

As described in Section 4.3, each application implemented using our software architecture is composed of a sequence of instructions from the APIPM instruction set. For global performance optimization it is not necessary to individually consider each of the instructions in such a sequence. Specific combinations of APIPM instructions often appear together, and are identical for sequential operation as well as for parallel execution. For such 'unbreakable' APIPM instruction sequences relating to sequential processing, we have introduced a shorthand notation, presented in Table 6.1.

Notation for unbreakable instruction streams relating to interprocess communication is given in Table 6.2. It contains abstractions similar to operations in MPI [104]. The additional `CreatLocalPart/Full` and `DelLocal` functions constitute creators and destructors for *partial* data structures (see Section 3.3.2). The `BorderExchange` function is as described in Section 5.3. Finally, the `Redistribute` function is included for completeness only, and implements a remapping of a distributed data structure onto a newly defined logical processor grid.

Partial structures are referred to as *local* in Table 6.2 (`locsrc` and `locdst`). The original data structure from which the partial data structures are obtained is referred to as *global* (`globsrc` and `globdst`). As an example, the `Scatter` operation requires a global source data structure as input, and produces the local (partial) destination structures as output, each of which is transferred to the node with the appropriate responsibility (see also Section 3.2).

For any application implemented using our software architecture it is possible to derive an abstract operation stream comprising of functions from Tables 6.1 and 6.2 alone. Consequently, in the remainder of this chapter we restrict our attention to abstract operation streams, and ignore the lower level APIPM instructions altogether.

```
Create      ( OUT  dst   );
Delete      ( OUT  dst   );
MemCopy     ( IN   src,  OUT  dst   );

UnPixOp     ( IN   src,  OUT  dst   );
BinPixOpV   ( IN   src,  OUT  dst,  IN  arg  );
BinPixOpI   ( IN   src,  OUT  dst,  IN  arg  );
ReduceOp    ( IN   src,  OUT  dst   );
NeighOp     ( IN   src,  OUT  dst,  IN  ker  );
GenConvOp   ( IN   src,  OUT  dst,  IN  ker  );
GeoMat      ( IN   src,  OUT  dst   );
GeoRoi      ( IN   src,  OUT  dst   );

Import      ( OUT  dst   );
Export      ( IN   src   );
```

Table 6.1: *Abstract function specifications for sequential operation (see Tables 4A.1 and 4A.2 for comparison).*

```
CreatLocalPart  (  OUT    locdst    );
CreatLocalFull  (  OUT    locdst    );
DelLocal        (  OUT    locdst    );

Broadcast       (  IN     globsrc,  OUT     locdst    );
Scatter         (  IN     globsrc,  OUT     locdst    );
Gather          (  IN     locsrc,   OUT     globdst   );
GatherAll       (  INOUT  locsrc,   INOUT   globdst   );
ReduceOne       (  INOUT  locsrc,   OUT     globdst   );
ReduceAll       (  INOUT  locsrc,   INOUT   globdst   );

BorderExchange  (  INOUT  locsrc    );
Redistribute    (  INOUT  locsrc    );
```

Table 6.2: *Additional abstract function specifications for parallel operation.*

6.1.2 Default Algorithm Expansion

In Section 3.4.2 we have indicated that all data structures applied in our library operations have a predefined *data access pattern type*. Each such type determines how accesses to non-local partial data structures are resolved with minimal communication overhead. From this information, a default approach for parallel execution directly follows for each library operation. The availability of a default parallelization strategy for each individual operation makes for a straightforward conversion of a complete sequential image processing application into an equivalent parallel program.

The conversion process, referred to as *default algorithm expansion*, is illustrated by the simple example code of Listing 6.1. The abstract sequential program, shown

```
Import( ImA );
UnPixOp( ImA, ImB );
BinPixOpI( ImB, ImC, ImA );
Export( ImC );
Delete( ImA );
Delete( ImB );
Delete( ImC );
```

(a) Sequential.

```
Import( ImA );
  Scatter( ImA, locImA );
UnPixOp( locImA, locImB );
  Gather( locImB, ImB );
  DelLocal( locImA );
  DelLocal( locImB );
  Scatter( ImA, locImA );
  Scatter( ImB, locImB );
BinPixOpI( locImB, locImC, locImA );
  Gather( locImC, ImC );
  DelLocal( locImA );
  DelLocal( locImB );
  DelLocal( locImC );
Export( ImC );
Delete( ImA );
Delete( ImB );
Delete( ImC );
```

(b) Parallel (default).

Listing 6.1: *Abstract sequential application (a) and equivalent parallel program after default algorithm expansion (b); additional operations in parallel code are indented.*

on the left, first imports image `ImA`, which is used as input to a unary pixel operation. Subsequently, the resulting output image `ImB` is used as input to a binary pixel operation. Finally, the resulting image `ImC` is exported, and all images are destroyed.

The equivalent parallel program, obtained after default algorithm expansion, is shown on the right of Listing 6.1. Because any data structure passed as input to a unary pixel operation is defined to have a one-to-one data access pattern type, a `Scatter` operation is inserted before the `UnPixOp` call. After the operation has finished, the resulting partial outputs are gathered to the single root node and all temporary partial data structures are destroyed. Subsequently, the images that are passed as source and argument to the binary pixel operation are spread throughout the parallel system in a `Scatter` operation. The partial outputs resulting from `BinPixOp` are gathered to the root, after which all partial structures are deleted. From this point onward, the program is identical to the original sequential version.

Default algorithm expansion in this manner is guaranteed to produce a legal and correct parallel version of any legal sequential program implemented using our software architecture. This is simply because each abstract function call in the sequential code is replaced by an equivalent sequence of one or more (parallel) operations. The resulting program is not guaranteed to be time-optimal, however. In fact, in most situations the expansion process will not even produce the fastest parallel implementation at all. Worse even, the resulting parallel code often can be expected to be *slower* than the original sequential program. Although other parallelization tools may be implemented differently, all library-based tools suffer from the very same problem — and for improved performance a solution is essential.

6.1.3 Inefficiencies from Default Algorithm Expansion

When considering the parallel code of Listing 6.1(b), it is clear that it contains several function calls that could be removed without violating the program's correctness or legality. First, image structure `locImA`, which is used as source structure for the unary pixel operation, is removed by `DelLocal` and subsequently recreated in the second occurrence of the `Scatter(ImA, locImA)` call. For improved performance, both operations simply could be removed. The same holds for the sequence of instructions applied to the `locImB` structure preceding the `BinPixOpI` call (i.e., `Gather` followed by `DelLocal` and `Scatter`). Listing 6.2(b) presents the optimized program obtained after removing the redundant communication steps from the parallel code.

A second source of inefficiencies is due to the fact that each individual communication step is performed irrespective of other message transfers in the program. Consequently, removal of redundant communication operations is a form of *local* performance optimization only, as it may be better to combine multiple messages in a single transfer. As an example, a `Scatter` operation followed by a `Broadcast` performed on the same data structure at a later stage in a program, could be replaced by a single `Broadcast` at the first essential point of message transfer, possibly followed by a `MemCopy` operation to extract the partial data structures on each processing unit.

A third category of performance inefficiencies is due to the fact that default algorithm expansion ignores the performance characteristics of the parallel machine at

```
Import( ImA );
UnPixOp( ImA, ImB );
BinPixOpI( ImB, ImC, ImA );
Export( ImC );
Delete( ImA );
Delete( ImB );
Delete( ImC );
```

(a) Sequential.

```
Import( ImA );
  Scatter( ImA, locImA );
UnPixOp( locImA, locImB );
BinPixOpI( locImB, locImC, locImA );
  Gather( locImC, ImC );
  DelLocal( locImA );
  DelLocal( locImB );
  DelLocal( locImC );
Export( ImC );
Delete( ImA );
Delete( ImB );
Delete( ImC );
```

(b) Parallel (optimized).

Listing 6.2: *Abstract sequential application (a) and equivalent parallel program after inter-operation optimization (b).*

hand. As indicated in Chapters 4 and 5, communication overhead also depends on the specifications of the underlying interconnection network, and the implementation of the applied message passing primitives. As a consequence, it is essential for the APIPM-based performance models of Chapters 4 and 5 to be incorporated in the optimization process as well.

From these types of inefficiencies, the first (i.e., the presence of redundancy) is by far the most important to be resolved. This is because redundant operations are responsible for the bulk of all unnecessary communication overhead. In fact, a program which is stripped of all redundant communication is generally quite efficient, and is often comparable to hand-optimized code. Redundancy avoidance is therefore the focal point of the optimization strategy proposed in the next sections. The latter two types of inefficiencies are still important, however, as these may have a significant impact on execution time as well. This is especially true for large clusters, as the relative impact of communication on performance increases with every node added to the system. Consequently, the remainder also proposes an extended optimization strategy that takes into account the latter two types of inefficiencies.

6.2 Finite State Machine Definition

To guide the process of operation removal, we have defined a *finite state machine* (*fsm*) which is used for operation redundancy detection, the monitoring of the life-span of (distributed) data structures, and the resolution of data structure inconsistencies. In this chapter, we restrict ourselves to so-called *deterministic finite acceptors*, which have no temporary storage and which can not produce strings of output. A deterministic finite acceptor (or *dfa*) is defined by the quintuple

$$M = (Q,\ \Sigma,\ \delta,\ q_0,\ F),$$

where

Q is a finite set of *internal states*,
Σ is a finite set of symbols called the *input alphabet*,
$\delta : Q \times \Sigma \rightarrow Q$ is a *transition function*,
$q_0 \in Q$ is the *initial state*,
$F \subseteq Q$ is a set of *final states*.

Initially, a dfa is assumed to be in the initial state q_0, with its input mechanism on the leftmost symbol of the input string. During each move of the automaton, one input symbol is consumed. When the end of the string is reached, the string is accepted if the automaton is in one of the final states. Otherwise, the string is rejected.

Deterministic finite acceptors as described here have been applied successfully in many fields of computer science, e.g. digital design, programming languages, and compilers [70, 96]. The following presents a specification of the finite state machine for global application optimization as applied in our software architecture.

6.2.1 States and Lifespan of (Distributed) Data Structures

As described in Section 3.3.2, for parallel execution two types of data structure representations are used in our software architecture: *global* structures and *local* (or partial) structures. A global structure always resides at a single processing unit (the root), and contains all data for the complete domain of the structure it represents. Local structures, on the other hand, are the result of a collective communication operation performed on a global structure.

There is a strong relationship between a global structure and the set of derived local structures (a set which is referred to as a *distributed data structure*). Clearly, at any time during the execution of a parallel program either the global structure itself or the distributed structure derived from that global structure must contain up-to-date values for all structure elements. An abstract representation of the relationship between these data structures is given by the three-tuple

$$q = (g,\ d,\ t),$$

where

$g \in G$ is the *state of the global structure*,
$d \in D$ is the *state of the derived distributed structure*,
$t \in T$ is the *distributed structure's distribution type*.

and

$G = \{$ `none, created, valid, invalid` $\}$,
$D = \{$ `none, valid, invalid` $\}$,
$T = \{$ `none, partial, full, not-reduced` $\}$.

In set G, `none` indicates that no space has been allocated for the global data structure in the main memory of the root. Furthermore, `created` indicates that space for the global structure has been allocated by way of the `Create` function. In

this state, the elements of the global structure do not contain values resulting from any calculation (yet). Finally, `valid` indicates that the global structure contains up-to-date values for all structure elements, and `invalid` indicates that at least one of the global structure's elements may contain an incorrect value. For distributed structures, the elements in set D are defined in a similar manner. The value `created` is not present in set D, however, simply because we do not need it.

In set T, `none` indicates that no distribution type information is available for the distributed structure. In addition, `partial` indicates that the set of constituent local structures is the result of a non-overlapping `Scatter` operation, while `full` indicates that the structures are obtained in a `Broadcast` operation. Finally, `not-reduced` indicates that all elements of the constituent, fully overlapping, local structures yet have to be subjected to an element-wise `ReduceOne` or `ReduceAll` operation.

The set $R = G \times D \times T$ contains all possible representations of the relationship between a global structure and its derived distributed structure. However, at application run time many of these possible representations can not (or should not) occur. As an example, a representation given by $q =$ `(invalid, invalid, full)` should not be present in a program, as neither the global structure nor the distributed structure contains all correct and up-to-date values. In addition, the representation given by $q =$ `(none, none, full)` is not useful, as it contains as much information as the more accurate representation $q =$ `(none, none, none)`.

For the finite state machine, we have specified a restricted set of *valid internal states*, based on the presented relationship between global and distributed structures. The selected set of valid internal fsm states is defined by

$$Q = \{\ q_0,\ q_1, \cdots,\ q_8\ \} \subset G \times D \times T,$$

with

$q_0 =$ `(none, none, none)`,
$q_1 =$ `(created, none, none)`,
$q_2 =$ `(valid, none, none)`,
$q_3 =$ `(invalid, none, none)`,
$q_4 =$ `(valid, valid, partial)`,
$q_5 =$ `(valid, valid, full)`,
$q_6 =$ `(invalid, valid, partial)`,
$q_7 =$ `(invalid, valid, full)`,
$q_8 =$ `(invalid, invalid, not-reduced)`.

State q_0 is the *empty state*, and represents the state of the global-distributed structure combination before its initial creation and after its final destruction. State q_1 represents the state immediately after creation of the global structure. This is a special case of state q_2, as the global structure also could be designated as `valid`. State q_1 is required to avoid communication in case a distributed structure is to be derived from a global structure in this state. State q_2 simply indicates that a global structure's elements contain all correct and up-to-date values, while a derived distributed structure is nonexistent. At first glance, q_3 seems to be a state that should never appear in a legal parallel program. However, this is the state obtained after performing a `DelLocal` operation in case the global-distributed structure combination is represented by states q_6, q_7, or q_8. In states q_4, q_5, q_6, and q_7, the distributed structure contains all correct values, while the related global structure is either consistent or inconsistent with

these values. Finally, state q_8 occurs in parallel reduction operations. As long as the required reduction has not yet been performed on the distributed structure, all constituent local structures as well as the related global structure remain invalid.

At run time each global-distributed structure combination starts in the empty state q_0. From this point onward each state can be reached, depending on the operations performed on the structure combination. Also, certain states can be reached multiple times. The *lifespan* of a global-distributed structure combination ends in case it returns to the empty state q_0. As such, state q_0 serves as the *initial state* of our finite state machine definition, as well as the single element in the set of *final states.*

6.2.2 State Transition Functions and State Dependencies

The *input alphabet* for our finite state machine is formed by the abstract functions of Tables 6.1 and 6.2, with a concrete data structure reference for each formal parameter. Also, as the fsm is used to monitor state changes and lifespan of a *single* data structure only, monitoring the correctness and legality of a complete application involves *multiple* finite state machines. The presence of multiple state machines results in a *parallel view* of the states of all data structures in an application. At any given moment during execution, several data structures are 'alive' and their combined state is captured by their respective finite state machines.

As the states of multiple data structures are not always *independent*, we assume that each fsm has a complete and up-to-date view of the states of all data structures in an application. Also, by way of the defined set of *state transition functions*, each state machine incorporates all knowledge regarding data structure state dependencies. To this end, the definition of state transition functions is extended as follows:

$$\delta : Q \times \Sigma_d \rightarrow Q,$$

where Σ_d is the input alphabet in which each (abstract) function is annotated with a list of permitted state dependencies for all additonal data structures passed as parameter to that function (i.e., those structures for which the current fsm is not responsible). Here, we represent elements in Σ_d by a two- or three-tuple, in which the first component is the name of the abstract function, and the remainder represents the (possibly empty) list of state dependencies. For example, $\delta(q_0, (\texttt{BinPixOpV}, q_4, q_5)) = q_6$ represents a state transition function for the output structure produced by the `BinPixOpV` operation. This transition function changes the state of the output structure from q_0 to q_6, while the source and argument structures are expected to be in states q_4 and q_5 respectively. It should be noted, that the knowledge obtained with this parallel view of state machines also could have been captured in a single *cross-product machine*, in which each deterministic finite automaton simulates, in parallel, the behavior of each component dfa (e.g., see [101]). For simplicity of notation, however, in the remainder of this chapter we keep to the parallel view of simple state machines.

Table 6.3 presents the state transition functions for the image processing functionality available in our software library. The overview is complete in the sense that our implementations allow no state transitions other than the ones presented here. In all cases, initial state q_0 refers to the state of the output structure produced by any of

the operations (represented by an OUT parameter in Table 6.1). As can be seen, output structures are the only structures that actually move from one state to another. Input structures and argument structures never change state, as these are accessed only, and never updated. All transition functions that cause a structure to be moved to state q_2 indicate fully sequential execution using global data structures only. All other transition functions refer to parallel execution using distributed data structures.

$$\delta(q_0, (\texttt{Create}, -)) = q_1, \qquad \delta(q_i, (\texttt{Delete}, -)) = q_0,$$
$$\delta(q_0, (\texttt{Import}, -)) = q_2, \qquad \delta(q_j, (\texttt{Export}, -)) = q_j,$$

with $i \in \{1, 2, 3\}, j \in \{1, 2, 4, 5\}$,

$$\delta(q_0, (\texttt{op}, q_2)) = q_2, \qquad \delta(q_0, (\texttt{op}, q_6)) = q_6,$$
$$\delta(q_0, (\texttt{op}, q_4)) = q_6, \qquad \delta(q_0, (\texttt{op}, q_7)) = q_7,$$
$$\delta(q_0, (\texttt{op}, q_5)) = q_7, \qquad \delta(q_i, (\texttt{op}, q_0)) = q_i,$$

with $\texttt{op} \in \{\texttt{Memcopy}, \texttt{UnPixOp}\}, i \in \{2, 4, 5, 6, 7\}$,

$$\delta(q_0, (\texttt{op}, q_2, q_2)) = q_2, \qquad \delta(q_2, (\texttt{op}, q_0, q_2)) = q_2,$$
$$\delta(q_0, (\texttt{op}, q_4, q_i)) = q_6, \qquad \delta(q_4, (\texttt{op}, q_0, q_i)) = q_4,$$
$$\delta(q_0, (\texttt{op}, q_5, q_i)) = q_7, \qquad \delta(q_5, (\texttt{op}, q_0, q_j)) = q_5,$$
$$\delta(q_0, (\texttt{op}, q_6, q_i)) = q_6, \qquad \delta(q_6, (\texttt{op}, q_0, q_i)) = q_6,$$
$$\delta(q_0, (\texttt{op}, q_7, q_i)) = q_7, \qquad \delta(q_7, (\texttt{op}, q_0, q_j)) = q_7,$$

with $\texttt{op} \in \{\texttt{BinPixOpV}, \texttt{NeighOp}, \texttt{GenConvOp}\}, i \in \{5, 7\}, j \in \{4, 5, 6, 7\}$,

$$\delta(q_0, (\texttt{BinPixOpI}, q_2, q_2)) = q_2, \qquad \delta(q_2, (\texttt{BinPixOpI}, q_0, q_2)) = q_2,$$
$$\delta(q_0, (\texttt{BinPixOpI}, q_i, q_j)) = q_6, \qquad \delta(q_i, (\texttt{BinPixOpI}, q_0, q_j)) = q_i,$$
$$\delta(q_0, (\texttt{BinPixOpI}, q_k, q_l)) = q_7, \qquad \delta(q_k, (\texttt{BinPixOpI}, q_0, q_l)) = q_k,$$

with $i, j \in \{4, 6\}, k, l \in \{5, 7\}$,

$$\delta(q_0, (\texttt{ReduceOp}, q_2)) = q_2, \qquad \delta(q_2, (\texttt{ReduceOp}, q_0)) = q_2,$$
$$\delta(q_0, (\texttt{ReduceOp}, q_i)) = q_8, \qquad \delta(q_i, (\texttt{ReduceOp}, q_0)) = q_i,$$
$$\delta(q_0, (\texttt{ReduceOp}, q_j)) = q_7, \qquad \delta(q_j, (\texttt{ReduceOp}, q_0)) = q_j,$$

with $i \in \{4, 6\}, j \in \{5, 7\}$,

$$\delta(q_0, (\texttt{op}, q_2)) = q_2, \qquad \delta(q_2, (\texttt{op}, q_0)) = q_2,$$
$$\delta(q_0, (\texttt{op}, q_i)) = q_6, \qquad \delta(q_i, (\texttt{op}, q_0)) = q_i,$$

with $\texttt{op} \in \{\texttt{GeoMat}, \texttt{GeoRoi}\}, i \in \{5, 7\}$.

Table 6.3: *State transition functions (including annotated state dependencies) for image processing functionality.*

$\delta(q_1, (\texttt{CreatLocalPart}, -)) = q_4,$	$\delta(q_i, (\texttt{DelLocal}, -)) = q_2,$
$\delta(q_1, (\texttt{CreatLocalFull}, -)) = q_5,$	$\delta(q_j, (\texttt{DelLocal}, -)) = q_3,$
with $i \in \{4, 5\}, j \in \{6, 7, 8\},$	
$\delta(q_2, (\texttt{Broadcast}, -)) = q_5,$	$\delta(q_8, (\texttt{ReduceOne}, -)) = q_2,$
$\delta(q_2, (\texttt{Scatter}, -)) = q_4,$	$\delta(q_8, (\texttt{ReduceAll}, -)) = q_5,$
$\delta(q_6, (\texttt{Gather}, -)) = q_4,$	$\delta(q_i, (\texttt{BorderExchange}, -)) = q_i,$
$\delta(q_7, (\texttt{Gather}, -)) = q_5,$	$\delta(q_i, (\texttt{Redistribute}, -)) = q_i,$
$\delta(q_6, (\texttt{GatherAll}, -)) = q_5,$	
with $i \in \{4, 6\}$.	

Table 6.4: *Additional state transition functions for parallel execution.*

State transition functions related to the additional communication functionality, and the memory management of local data structures, are presented in Table 6.4. In all of these transition functions the list of state dependencies is empty, as the functions work on a single data structure only. The importance of the additional transition functions is that these are used to resolve data structure state inconsistencies which may appear in an application. As an example, consider the first three lines of code in Listing 6.1(b). The first operation (`Import`) moves structure `ImA` from q_0 to q_2 (see Table 6.3). In case the third operation (`UnPixOp`) is to be executed in parallel, the input data structure is expected to be in one of the states q_4, q_5, q_6, or q_7. None of these states immediately matches with the output state of structure `ImA` after the `Import` operation. This state inconsistency is resolved by executing a `Scatter` operation (as in Listing 6.1(b)) or a `Broadcast` operation immediately after the `Import` operation. This is because these operations change an input structure's state from q_2 either to q_4, or to q_5 respectively (see Table 6.4).

Figure 6.1 presents a reduced state transition graph for our finite state machine definition. For better readability, the graph contains only those operations that actually cause a data structure to move from one state to another state. As such, the graph incorporates the complete lifespan of a data structure, and covers any state a data structure can possibly reach at run time. Also, it should be noted that it is exactly these operations that play an essential role in the process of operation redundancy avoidance as will be presented in Section 6.3.

6.2.3 Legal Sequential Code and Legal Parallel Code

A program is a *legal program*, if and only if it is accepted by *all* finite state machines related to that program. In other words, a program is legal if (1) it consists of a sequence of abstract function calls from Tables 6.1 and 6.2 only, (2) it contains no data structure state inconsistencies, and (3) all internal data structures start as well as

end in the empty state q_0. In case a user-provided sequential program is accepted as a legal program, the process of default algorithm expansion always generates a legal and correct parallel program as well. This is because each sequence of (parallel) operations that replaces a sequential call generates exactly the same set of data structure state transitions at all times. The following section shows how the presented finite state machine definition is used to obtain legal and correct parallel code, which is optimized in that the execution of any redundant communication operations is avoided.

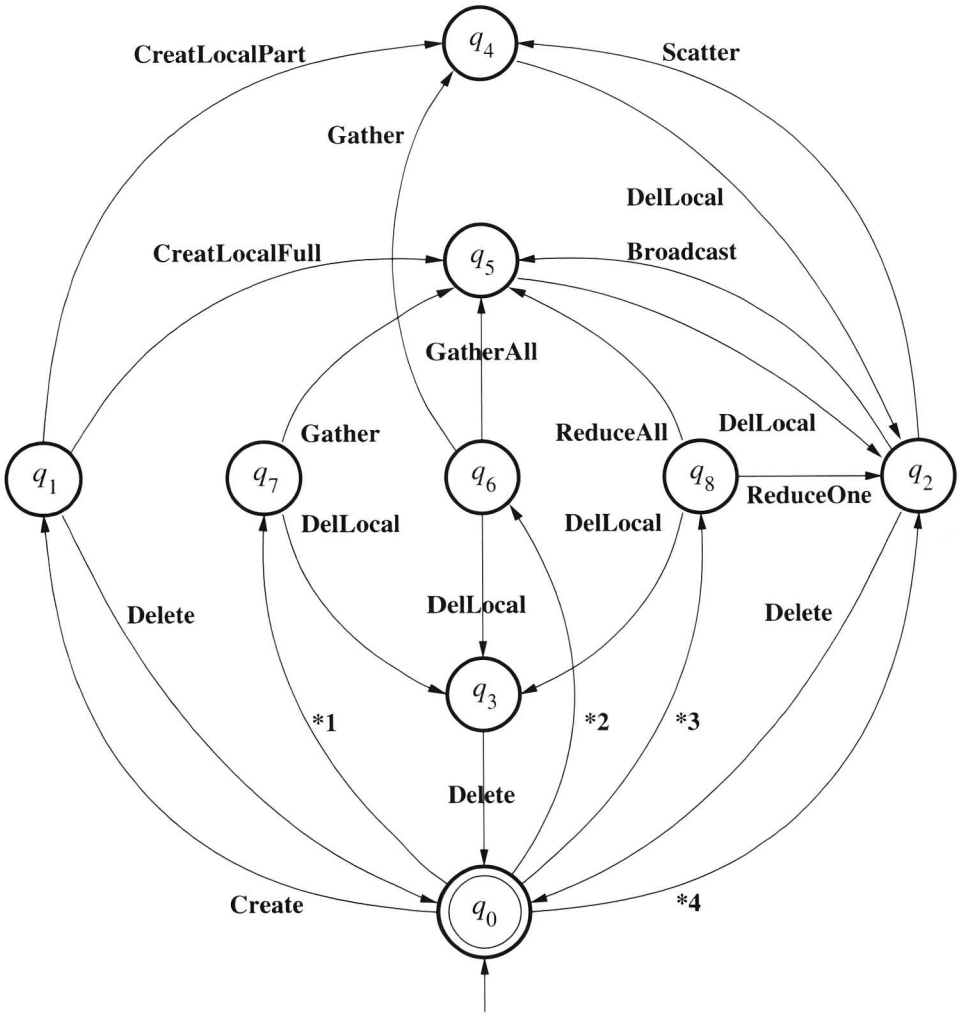


Figure 6.1: *Reduced state transition graph.*

6.3 Redundancy Avoidance by Lazy Parallelization

In the approach of *lazy parallelization* it is simply assumed that *each* communication or memory management operation inserted in the default algorithm expansion process is redundant, unless proven otherwise. Stated differently, lazy parallelization causes a default communication or memory management operation to be executed only, in case its removal would introduce an (immediate) data structure state inconsistency. Although lazy parallelization can be applied on the fly at run time, for the moment we will present it as a compile time method. Conceptually, the approach of lazy parallelization consists of the following parallelization and optimization steps:

1. Apply the process of default algorithm expansion to the original sequential code.
2. Scan the expanded code, and remove *all* communication operations, as well as *all* operations for the creation and destruction of partial data structures.
3. Apply *partial loop unrolling* by extracting the code for the first iteration of each loop, and placing it in front of the code for the remaining loop iterations.
4. Resolve all introduced data structure state inconsistencies by re-inserting operations removed in step 2.
5. Undo the partial loop unrolling by replacing all separated loops by a single combined code block.

As stated, the code obtained after the first step consists of legal, but non-optimal parallel code. The operation removal in the second step, however, introduces many state inconsistencies. These are resolved in step four. As will be described below, in this step any illegal parallel code is transformed to legal code by (re-)inserting operations to resolve data structure state inconsistencies. Steps 3 and 5 are present only to deal with loop constructs which may be present in the user-provided code. The extraction of the first iteration of a loop (*partial loop unrolling*) exposes all data structure state inconsistencies that can possibly occur in a program. More specifically, loop unrolling makes it possible to compare (1) the data structure states reached after execution of the pre-loop code with the states required in the first loop iteration, (2) the states reached after execution of the n-th loop iteration with the states required in iteration $n + 1$, and (3) the states reached after execution of the last loop iteration with the states required in the post-loop code.

Listing 6.3 gives an example of the application of lazy parallelization. The abstract code for a simple example program is shown in Listing 6.3(a). The programs obtained in the first three steps of the optimization process are all straightforward, and will not be explained any further. The re-insertion of code as applied in step 4 (see Listing 6.3(e)) is performed using the state transition functions of Section 6.2.2 (i.e., only those incorporated in the reduced state transition graph of Figure 6.1). The `Broadcast( ImA, locImA )` operation in the first loop iteration is inserted because the `Import` operation causes its output structure to be moved to state q_2, while for parallel execution the subsequent `GeoMat` operation requires its input structure to

```
Import( ImA );
LOOP [1:N]
   GeoMat( ImA, ImB );
   GenConvOp( ImB, ImC, k );
   Export( ImC );
   Delete( ImC );
   Delete( ImB );
ENDLOOP
Delete( ImA );
```

(a) example sequential code

```
Import( ImA );
LOOP [1:N]
   Broadcast( ImA, locImA );
   GeoMat( locImA, locImB );
   Gather( locImB, ImB );
   DelLocal( locImB );
   DelLocal( locImA );
   Scatter( ImB, locImB );
   GenConvOp( locImB, locImC, k );
   Gather( locImC, ImC );
   DelLocal( locImC );
   DelLocal( locImB );
   Export( ImC );
   Delete( ImC );
   Delete( ImB );
ENDLOOP
Delete( ImA );
```

(b) after step 1

```
Import( ImA );
LOOP [1:N]
   GeoMat( locImA, locImB );
   GenConvOp( locImB, locImC, k );
   Export( ImC );
   Delete( ImC );
   Delete( ImB );
ENDLOOP
Delete( ImA );
```

(c) after step 2

```
Import( ImA );
LOOP [1]
   GeoMat( locImA, locImB );
   GenConvOp( locImB, locImC, k );
   Export( ImC );
   Delete( ImC );
   Delete( ImB );
ENDLOOP
LOOP [2:N]
   GeoMat( locImA, locImB );
   GenConvOp( locImB, locImC, k );
   Export( ImC );
   Delete( ImC );
   Delete( ImB );
ENDLOOP
Delete( ImA );
```

(d) after step 3

```
Import( ImA );
LOOP [1]
   Broadcast( ImA, locImA );
   GeoMat( locImA, locImB );
   GenConvOp( locImB, locImC, k );
   Gather( locImC, ImC );
   Export( ImC );
   DelLocal( locImC );
   Delete( ImC );
   DelLocal( locImB );
   Delete( ImB );
ENDLOOP
LOOP [2:N]
   GeoMat( locImA, locImB );
   GenConvOp( locImB, locImC, k );
   Gather( locImC, ImC );
   Export( ImC );
   DelLocal( locImC );
   Delete( ImC );
   DelLocal( locImB );
   Delete( ImB );
ENDLOOP
DelLocal( locImA );
Delete( ImA );
```

(e) after step 4

```
Import( ImA );
LOOP [1:N]
   IF [1] Broadcast( ImA, locImA);
   GeoMat( locImA, locImB );
   GenConvOp( locImB, locImC, k );
   Gather( locImC, ImC );
   Export( ImC );
   DelLocal( locImC );
   Delete( ImC );
   DelLocal( locImB );
   Delete( ImB );
ENDLOOP
DelLocal( locImA );
Delete( ImA );
```

(f) after step 5

Listing 6.3: *Example of code optimization by lazy parallelization (compile time): (a) original sequential code, (b) code obtained after default algorithm expansion, (c) code obtained after removal of 'redundant' communication operations and memory management operations, (d) code obtained after partial loop unrolling, (e) code obtained after resolution of state inconsistencies by default operation re-insertion, (f) optimized parallel code obtained after loop recombination.*

be in state q_5 or q_7 (see Table 6.3). The only available operation that provides a resolution to this state inconsistency is the `Broadcast` operation, as it moves a data structure from state q_2 to q_5. Similarly, `Gather( locC, C )` is inserted in the first loop iteration, as it moves `C` from q_6 to q_4, which is one of the allowed input states for the subsequent `Export` operation. The additional operation re-insertions work in a similar manner, and all further interpretation of Listing 6.3 is left to the reader.

6.3.1 Discussion

Lazy parallelization produces legal and correct parallel code at all times. This can be seen by considering the allowed states for all data structures passed as parameters to the operations in Table 6.1, and the resulting states for the output structures produced by these operations. As such, each operation has a set of *allowed input states* for each of its parameters, and one of these is moved to a new *output state*. By exhaustion, it is easily shown that for each possible output state, a sequence of zero or more state transition functions exists that moves a data structure from that particular output state to one state in each set of allowed input states.

An important property of the approach is that it can be applied on the fly at run time (hence its name). Because the required data structure states are known for each operation, it is possible to defer decisions regarding the execution of each default communication operation or memory management operation to as late as the actual moment of intended execution. Essentially, this means that all five steps as described above, are reduced to a single step. As such, lazy parallelization is unrestrictive and highly efficient, as no prior knowledge regarding the behavior of loops and branches in the code is required. This knowledge is simply obtained during execution of the application, and is not required any sooner.

Although lazy parallelization works well in many situations, it does not guarantee to always produce the *fastest possible* version of a program under consideration. First, this is because the approach always applies the fastest communication step whenever message transfer is mandatory. This is a form of *local* performance optimization, however, as it may be better to insert a *combined* message transfer to avoid further communication steps to be executed at a later stage. Secondly, the approach does not incorporate any knowledge obtained from our APIPM-based performance models described in Chapters 4 and 5. To overcome these problems, the next section proposes an extension to the approach of lazy parallelization, such that it is indeed capable of producing the (expected) fastest parallel version of any sequential program.

6.4 Application State Transition Graph

The process of lazy parallelization always results in the execution of a single pre-selected solution for resolution of data structure state inconsistencies. For each specific state inconsistency, each default resolution represents the cheapest operation (or sequence of operations) that is available in the software library. Although this strategy generally produces parallel code which is quite efficient, the approach is sub-optimal,

as it does not acknowledge that

1. the execution of more costly communication steps (e.g., `Broadcast` instead of `Scatter`) may avoid additional communication at a later stage in the program,
2. a single straightforward domain decomposition may deliver non-optimal performance (see Chapter 5),
3. the optimal routing pattern for the distribution of data partially depends on the characteristics of the interconnection network (again, see Chapter 5), and
4. the use of *all* available processing power is not always time-optimal.

Optimization in the light of these issues is obtained by constructing an *application state transition graph* (or *ASTG*), that characterizes an application's run time behavior, and incorporates all possible (combinations of) parallelization and optimization solutions. By annotating the vertices in the graph (representing all operations which are possibly performed by the application) with cost estimations obtained from our APIPM-based performance models described in Chapters 4 and 5, the expected optimal parallel implementation for an application is represented by the cheapest branch.

Figure 6.2 shows a simplified version of the ASTG constructed for the first three lines of code in Listing 6.1, assuming that a maximum number of only two processing units is available. After execution of the `Import` operation, several different execution paths can be followed. One choice is to execute the `UnPixOp` in a sequential manner, as is depicted by the uppermost branch in the graph. Parallel solutions involve either a `Scatter` operation or a `Broadcast` operation performed on the imported data structure. As explained in the Chapter 5, multiple versions of these operations exist in our

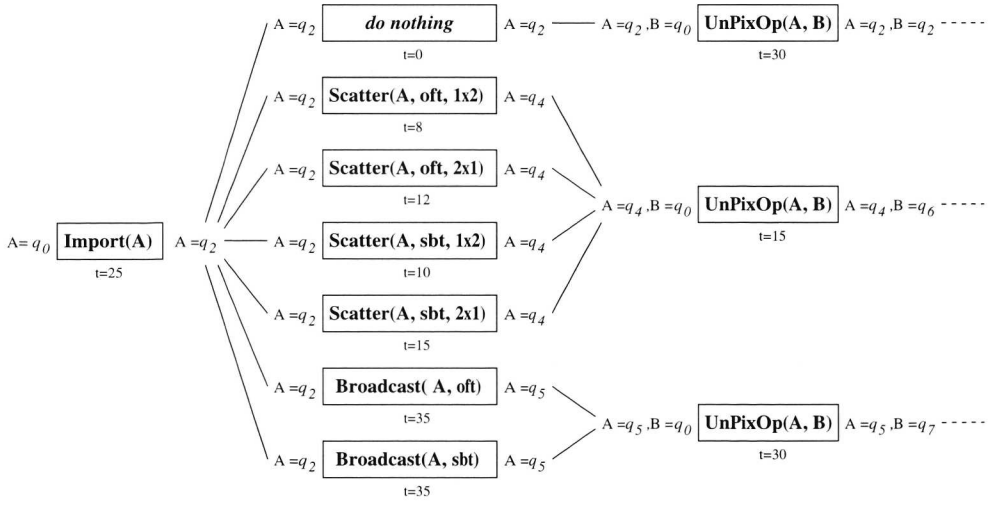


Figure 6.2: *Simplified partial application state transition graph.*

library, each having different performance characteristics. For the `Scatter` operation, it is required to also choose a logical processor grid onto which the data structure is to be mapped (see Section 3.2). All of these choices result in a different expected execution time for the program, as is indicated by the annotated performance estimations at each vertex in the graph. Although one of the branches in Figure 6.2 is cheapest for this initial part of the program, to obtain optimal performance for the complete application a different path may have to be followed.

Discussion

While the expected optimal parallel implementation is always obtained in this manner, the construction of a complete ASTG has several major disadvantages. First, in order to find the cheapest branch, the creation of an ASTG needs to be performed at compile time. As such, the approach is restrictive, as it is now required to have prior knowledge regarding the branching behavior of the application at hand. Another drawback is that it is often costly to actually obtain the cheapest branch in the graph. This is because an ASTG is generally large, even for applications of moderate size.

6.4.1 Heuristics for Search Space Reduction

To overcome the stated problems, we have defined several heuristics to reduce the size of any application state transition graph. The use of heuristics implies that our approach can no longer guarantee to find the expected optimal parallel implementation for any sequential program. However, in almost all situations a close-to-optimal program is still obtained, and application performance is generally still comparable to that of optimal hand-crafted parallel code (as will be demonstrated for all example applications evaluated in Chapter 7).

First, to overcome the problem of having to acquire prior knowledge regarding the branching behavior of an application, our optimization approach simply ignores unknown branches. At run time, any code block that has not been evaluated because of undetermined conditional behavior is simply executed according to the default lazy parallelization approach. In such a situation, all current logical data structure mappings are maintained, however, to avoid having to execute costly remapping operations. Although this approach solves the problem in the simplest possible way, it should be noted that we have learned that not many applications implemented using our software architecture actually contain such unknown branches.

A significant reduction of any ASTG is obtained by assuming that a specific data mapping that was found to be optimal for a certain operation, is also optimal for other operations with similar behavior. In other words, it is simply assumed that each parallelizable pattern entails a single optimal data partitioning strategy, irrespective of the actual operation that is implemented by that pattern. As a consequence, in an ASTG a sequence of operations applied to the same set of data structures is often reduced to a single block of code which is not 'interrupted' by any communication operations. Once a data structure has been partitioned and distributed, its logical mapping is maintained as long as possible.

An ASTG is also significantly reduced by assuming that data structures that are used as arguments representing kernel structures (as in the `GenConvOp` operation) or vector data (as in the `BinPixOpV` operation) at *any* point in a program, are never to be partitioned. This is realistic, as such data structures are usually much smaller than regular image data structures. Calculations on such small structures are simply assumed not to gain from parallel execution at all.

Other heuristics, such as evaluating partial execution paths either for a single node or for all available nodes only, and considering only a small number of possible logical data mappings for the maximum system size, also reduce each ASTG significantly. It is expected that additional heuristics can reduce each ASTG even further, without compromising too much on the run time performance of the resulting parallel code. This, however, is research we have left as future work.

6.5 Related Work

Although a multitude of library-based environments has been described in the literature, the process of optimization across library calls is not explicitly incorporated in many of these. Even in several relatively recent software architectures, performance optimization issues often are considered at the intra-operation level only (e.g., see [80, 81, 86, 87, 93, 111, 154, 159]). Other environments (e.g., [118]) leave part of the optimization process to a third-party compiler, as these require applications to be implemented in a high-level parallel language such as Compositional C++ [26].

The environment implemented by Morrow et al. [109] does incorporate a partial mechanism for inter-operation optimization. It is based on the concept of a *self-optimizing class library*, which is extended automatically with optimized parallel operations. In case a program is being executed for the first time, a syntax graph is constructed for each statement in the program, which is evaluated when an assignment operator is met. Any such syntax graph for combinations of primitive instructions (i.e., those incorporated as a single routine within the library) is written out for later consideration by an off-line optimizer. On subsequent runs of the program, a check is made to decide if an optimized routine is available for a given sequence of library calls. Although optimal performance may be guaranteed for a sequence of library routines in this manner, a drawback of this approach is that time-optimality is often not obtained for complete applications.

Other environments, such as developed by Jamieson et al. [73, 74], Lee et al. [94, 95], and Moore et al. [108], do incorporate a method for full inter-operation optimization. In all of these architectures the methods are purely static, however, and can be applied at compile time only. In this respect, our approach of lazy parallelization is more flexible, as it allows much of the optimization process to be performed at run time – without any significant overhead cost. In case run time performance obtained from the standard lazy parallelization approach is deemed insufficient, one can decide to incorporate additional compile time results obtained from the ASTG.

As far as we know, the use of a finite state machine specification is new in the field of library-based parallel imaging environments. Moreover, to our knowledge the

application of an fsm definition has not been considered at all in the field of parallel image processing. In several related research areas, however, fsm definitions have been applied before. For example, Chatterjee et al. [27] apply a finite state machine for the generation of optimal communication sets in distributed-memory implementations of data-parallel languages such as High Performance Fortran. As in our case, results indicate that the fsm approach requires very little runtime overhead. For ad-hoc optimization of specific algorithms (e.g., see [31]), or complete applications (e.g., see [106]), finite state machine definitions have been applied successfully as well.

Interestingly, our approach to finding optimal performance of operations as well as complete applications is related to several projects in other domains. The SPIRAL project [99, 152], for example, is aimed at the design of a system to generate efficient libraries for digital signal processing algorithms. SPIRAL generates efficient implementations of algorithms expressed in a domain-specific language, called SPL, by a systematic search through the space of possible implementations. Other efforts in automatically generating efficient implementations of programs include FFTW [51] for adaptively generating time-optimal FFT algorithms, and the ATLAS project [169] for deriving efficient implementations of basic linear algebra routines.

Finally, our work shares common goals with that of Baumgartner et. al. [14], in the search of an optimal data partitioning strategy with minimal communication overhead for applications in the field of quantum chemistry and physics. Similar to our work, an operator tree is generated, in which multiple data partitioning and communication strategies are incorporated. This work goes even one step further, in that memory usage is to be optimized at the same time. This approach is also entirely static, however, and includes no possibility for partial optimization performed at run time.

6.6 Conclusions

In this chapter we have presented a finite state machine based approach for global optimization of data parallel image processing applications. The approach, called lazy parallelization, considers a sequential program, which is parallelized automatically by inserting communication operations and local memory management operations whenever necessary. The approach generates legal, correct, and efficient parallel programs, given any sequential program implemented using our software architecture.

The main advantage of the optimization approach is that it can be applied on the fly at run time. As a result, the primary importance of lazy parallelization over other approaches described in the literature lies in the fact that it requires no a priori knowledge regarding the branching behavior of the application at hand. An additional advantage of lazy parallelization is that it requires very little runtime overhead. Also, in our software architecture it proved to be possible to incorporate the approach in an elegant manner — i.e., such that the long-term sustainability of the library implementations is not compromised.

Although lazy parallelization was shown to work well in many situations, it can not guarantee to always produce the fastest possible version of the program under consideration. To overcome this problem, an extension to the approach of lazy par-

allelization was also presented. The extended technique requires an application state transition graph (ASTG) to be generated. An ASTG incorporates all optimization decisions which can possibly be made at application run time. As each decision is annotated with a run time cost estimation (obtained from our APIPM-based performance models), the fastest version of the program is represented by the 'cheapest' branch in the ASTG.

An important drawback of the application state transition graph, however, is that it is often costly to actually obtain the cheapest branch. This is because the ASTG is generally large, even for applications of moderate size. For this reason we have also defined additional heuristics for search space reduction. Another drawback is that the creation and traversal of an ASTG can not be performed at run time. However, in case the default approach of lazy parallelization proves to deliver sufficiently high performance, the creation of an ASTG can be avoided altogether.

In conclusion, lazy parallelization on the basis of a finite state machine specification has proven to constitute a surprisingly simple, yet effective method for global optimization of data parallel image processing applications. Essentially, the simplicity stems from the high level abstractions incorporated in the fsm definition. Consequently, we feel that a similar approach could be applicable in other library-based architectures as well. This is especially true for the many environments for linear algebra operations, which include similar patterns of communication and calculation.

Chapter 7

Efficient Applications in User Transparent Parallel Image Processing*

"Thy will by my performance shall be serv'd:
So make the choice of thy own time, for I,
Thy resolv'd patient, on thee still rely."

William Shakespeare - *All's Well That Ends Well* (1623)

In the previous chapters we have described the essential and most innovative aspects of our software architecture for user transparent parallel image processing. First, in Chapter 2 we have discussed the need for the availability of such architecture, and we have presented a bird's eye view of all of the architecture's constituent components. In Chapter 3 we have presented some of the implementation details of the architecture's core — which is a sustainable software library consisting of an extensive set of operations commonly applied in state-of-the-art image processing research. In Chapter 4 we have introduced a performance model, derived from a high level abstract parallel image processing machine definition, which is used for obtaining accurate run time cost estimations for all operations available in our architecture. In addition, in Chapter 5 we have presented an extended model for accurate prediction of the cost of the basic point-to-point communication operations applied in the library implementations. As discussed in Chapter 6, performance estimations obtained from these models are essential for generating the fastest possible parallel version of any

*This chapter is based on our paper as appeared in *Proceedings of the 16th International Parallel & Distributed Processing Symposium (IPDPS 2002)* [150]. An extended version of this chapter is to appear in *Concurrency and Computation: Practice and Experience* [146].

sequential program implemented using our software architecture. In relation to this, in Chapter 6 we have also presented a finite state machine specification, which is used for the automatic conversion of a legal sequential image processing application into a legal, correct, and time-optimal parallel version of the same program.

For each of the research issues presented in the previous chapters, we have discussed the advantages and drawbacks of the solutions incorporated in our software architecture. Where possible, we have also presented results for each of the solutions applied in isolation. To validate *all* of the results of this research, however, the single remaining issue that has yet to be discussed in this thesis is the overall efficiency obtained in case the architecture components are applied in combination.

To this end, in this chapter we give an assessment of the software architecture's effectiveness in providing significant performance gains. In particular, we describe the implementation and automatic parallelization of three well-known example applications that contain many operations commonly applied in image processing research: (1) template matching, (2) multi-baseline stereo vision, and (3) line detection. For all three applications we determine whether the performance obtained with the parallel versions generated by our software architecture indeed adheres to requirement I.2 put forward in Section 2.3 which states that the obtained efficiency generally should compare well to that of reasonable hand-coded parallel implementations.

This chapter is organized as follows. First, in Section 7.1 we give a short description of the hardware architecture that we have used for all evaluation purposes. Next, in each of the Sections 7.2, 7.3, and 7.4, one of the example applications is described and evaluated in extensive detail. Information regarding the parallelization and optimization issues of each application is presented, in combination with obtained performance results and speedup characteristics. Where available, measurement data presented in the literature are compared with performance results obtained with our software architecture. Finally, concluding remarks are given in Section 7.5.

7.1 Hardware Environment

All of the applications described in this chapter have been implemented and tested on the 128-node homogeneous Distributed ASCI Supercomputer (DAS) cluster located at the Vrije Universiteit in Amsterdam [7]. This is a typical example of a machine from the class of homogeneous commodity clusters as described in Section 2.1. All nodes in the cluster contain a 200 Mhz Pentium Pro with 128 MByte of EDO-RAM, and are connected by a 1.2 Gbit/sec full-duplex Myrinet SAN network. At the time of measurement, the nodes ran the RedHat Linux 6.2 operating system. The software architecture was compiled using `gcc` 3.0 (at highest level of optimization) and linked with MPI-LFC [16], an implementation of MPI which is partially optimized for the DAS. The required set of benchmarking operations (see Section 4.4) was run on a total of three DAS nodes, under identical circumstances as the complete software architecture itself. At the time of measurement, 8 nodes in the DAS cluster were unusable due to a malfunction in the related network cards. As a consequence, performance results are presented for a system of up to 120 nodes only.

7.2 Template Matching

Template matching is one of the most fundamental tasks in many image processing applications. It is a simple method for locating specific objects within an image, where the template (which is, in fact, an image itself) contains the object one is searching for. For each possible position in the image the template is compared with the actual image data in order to find subimages that match the template. To reduce the impact of possible noise and distortion in the image, a similarity or error measure is used to determine how well the template compares with the image data. A match occurs when the error measure is below a certain predefined threshold.

In the example application described here, a large set of electrical engineering drawings is matched against a set of templates representing electrical components, such as transistors, diodes, etc. Although more post-processing tasks may be required for a truly realistic application (such as obtaining the actual positions where a match has occurred), we focus on the template matching task, as it is by far the most time-consuming. This is especially so because, in this example, for each input image f error image ε is obtained by using an additional *weight* template w to put more emphasis on the characteristic details of each 'symbol' template g:

$$\varepsilon(i,j) = \Sigma_m \Sigma_n ((f(i+m, j+n) - g(m,n))^2 \cdot w(m,n)). \tag{7.1}$$

When ignoring constant term $g^2 w$, this can be rewritten as:

$$\varepsilon = f^2 \otimes w - 2 \cdot (f \otimes w \cdot g). \tag{7.2}$$

with $\otimes$ the convolution operation. The error image is normalized such that an error of zero indicates a perfect match and an error of one a complete mismatch. Although the same result can be obtained using the Fast Fourier Transform (which has a better theoretical run time complexity, and also provides immediate localization of the best match and all of its resembling competitors), this brute force method is fastest for our particular data set.

7.2.1 Sequential Implementation

Listing 7.1 is a sequential pseudo code representation of Equation (7.2). The library calls are as described in Chapter 3. Essentially, each input image is read from file, squared (to obtain f^2), and matched against all symbol and weight templates, which are also obtained from file. In the inner loop the two convolution operations are performed, and the error image is calculated and written out to file.

7.2.2 Parallel Execution

As all parallelization issues are shielded from the user, the pseudo code of Listing 7.1 directly constitutes a program that can be executed in parallel as well. Efficiency of parallel execution depends on the optimizations performed by the architecture's scheduling component. For this particular sequential implementation, the optimization process (as described in Chapter 6) has generated a schedule that requires only

```
FOR i=0:NrImages-1 DO
  InputIm = ReadFile(...);
  SqrdInputIm = BinPixOp(InputIm, "mul", InputIm);
  FOR j=0:NrSymbols-1 DO
    IF (i==0) THEN
      weights[j] = ReadFile(...);
      symbols[j] = ReadFile(...);
      symbols[j] = BinPixOp(symbols[j], "mul", weights[j]);
    FI
    FiltIm1 = GenConvOp(SqrdInputIm, "mult", "add", weights[j]);
    FiltIm2 = GenConvOp(InPutIm, "mult", "add", symbols[j]);
    FiltIm2 = BinPixOp(FiltIm2, "mult", 2);
    ErrorIm = UnPixOp(FiltIm1, "sub", FiltIm2);
    WriteFile(ErrorIm);
  OD
OD
```

Listing 7.1: *Pseudo code for template matching.*

four different communication steps to be executed. First, each input image read from file is scattered throughout the parallel system (generally applying a logical CPU grid of $2 \times (P \div 2)$ or $4 \times (P \div 4)$, depending on the available number of nodes P). Next, in the inner loop all templates are broadcast to all processing units. Also, in order for the convolution operations to perform correctly, image borders (or *shadow regions*) are exchanged among neighboring nodes in the logical CPU grid. In all cases, the extent of the border in each dimension is half the size of the template minus one pixel. Finally, before each error image is written out to file it is gathered to a single processing unit. Apart from these communication operations all processing units can run independently, in a fully data parallel manner. As such, the program executes in exactly the same way as would have been the case for a hand-coded parallel version.

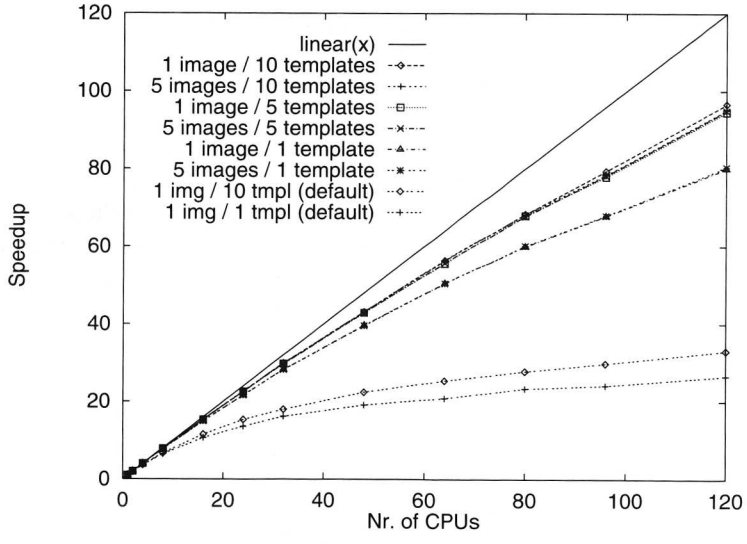
7.2.3 Performance Evaluation

Because template matching is such an important task in image processing, it is essential for our software architecture to perform well for this application. The results obtained for the automatically optimized parallel version of the program, presented in the first six columns of Figure 7.1(a), show that this is indeed the case. For these results, the graph of Figure 7.1(b) shows that even for a large number of processing units, speedup is close to linear. As was to be expected, the speedup characteristics are identical when the same number of templates is used in the matching process, irrespective of the number of input images.

It should be noted that the '1 template' case represents a lower bound on the obtainable speedup (which is slightly over 80 for 120 nodes). This is because in this situation the communication versus computation ratio is highest for the presented parallel system sizes. Additional measurements have indicated that the '10 template' case is a representative upper bound (with a speedup of more than 96 for 120 nodes). Even when up to 50 templates are being used in the matching process, the speedup characteristics were found to be almost identical to this upper bound.

# CPUs	time– optimized parallel program						default parallel program	
	1 input image			5 input images			1 input image	
	1 template (s)	5 templates (s)	10 templates (s)	1 template (s)	5 templates (s)	10 templates (s)	1 template (s)	10 templates (s)
1	25.439	126.654	253.165	127.158	632.485	1265.425	25.526	253.627
2	12.774	63.410	126.694	63.819	316.921	633.083	13.466	133.443
4	6.449	31.895	63.707	32.237	159.497	318.559	7.126	69.924
8	3.287	16.138	32.212	16.435	80.655	161.303	3.972	37.975
16	1.703	8.254	16.459	8.519	41.263	82.259	2.399	21.960
24	1.176	5.618	11.207	5.876	28.078	55.838	1.876	16.539
32	0.902	4.261	8.473	4.508	21.318	42.414	1.581	14.128
48	0.642	2.956	5.875	3.218	14.751	29.367	1.337	11.330
64	0.503	2.280	4.493	2.523	11.353	22.409	1.224	9.998
80	0.424	1.865	3.708	2.115	9.340	18.546	1.093	9.119
96	0.375	1.627	3.189	1.871	8.088	16.146	1.056	8.493
120	0.317	1.340	2.619	1.581	6.659	13.299	0.960	7.668

(a)



(b)

Figure 7.1: *Performance and speedup characteristics for template matching using input images of 1093×649 (4-byte) pixels and templates of size 41×35. (a) Execution times in seconds for multiple combinations of templates and images. Results in first six columns obtained for optimized parallel version. Results in last two columns (gray) obtained for non-optimized parallel version generated by default algorithm expansion. (b) Speedup graph for all measurements. Four uppermost lines for optimized program calculating matches for 5 and 10 templates; two lower lines for matching with a single template. Bottom two lines for non-optimized (default) parallel program.*

The additional values in the gray columns of Figure 7.1(a) represent measurement results obtained for a non-optimized parallel version of the program (i.e., the parallel program which is obtained in the process of default algorithm expansion, without applying a redundancy avoidance strategy or any other optimization steps, see Section 6.1.2). These measurements, as well as the related speedup characteristics shown in Figure 7.1(b), clearly indicate the importance of the optimization process presented in Chapter 6. Most importantly, the dramatic results are due to the fact that the default parallel program executes many redundant communication steps. For evaluation of the efficiency of our software architecture, these non-optimized results simply should be ignored. In the remainder of this chapter we will therefore only present results for time-optimized parallel programs.

7.3 Multi-Baseline Stereo Vision

As indicated in [82, 110], depth maps obtained by conventional stereo ranging, which uses correspondences between images obtained from two cameras placed at a small distance from each other, are generally not very accurate. In part, this is due to the fundamental difficulty of the stereo correspondence problem: finding corresponding points between left and right images is locally ambiguous. Several solutions to this problem have been proposed in the literature, ranging from a hierarchical smoothing or coarse-to-fine strategy, to a global optimization technique based on surface coherence assumptions. These techniques, however, tend to be heuristic or result in computationally expensive algorithms.

In [117], Okutomi and Kanade propose an efficient *multi-baseline stereo vision* method, which is more accurate for depth estimation than more conventional approaches. Whereas, in ordinary stereo, depth is estimated by calculating the error between two images, multi-baseline stereo requires more than two equally spaced

(a) (b)

Figure 7.2: *Example of typical input scene (a) and extracted depth map (b). Courtesy of Professor H. Yang, University of Alberta, Canada.*

cameras along a single *baseline* to obtain redundant information. In comparison with two-camera methods, multi-baseline stereo was shown to significantly reduce the number of false matches, thus making depth estimation much more robust.

In the algorithm discussed here, input consists of images acquired from three cameras. One image is the *reference* image, the other two are *match* images. For each of 16 disparities, $d = 0, \cdots, 15$, the first match image is shifted by d pixels, the second image is shifted by $2d$ pixels. First, a *difference* image is formed by computing the sum of squared differences between the corresponding pixels of the reference image and the shifted match images. Next, an *error* image is formed by replacing each pixel with the sum of the pixels in a surrounding 13×13 window. The resulting *disparity* image is then formed by finding, for each pixel, the disparity that minimizes the error. The depth of each pixel then can be displayed as a simple function of its disparity. A typical example of a depth map extracted in this manner is given in Figure 7.2.

7.3.1 Sequential Implementations

The sequential implementation used in this evaluation is based on a previous implementation written in a specialized parallel image processing language, called Adapt [166] (see also Section 2.2.2). As shown in Listing 7.2, for each displacement two disparity images are obtained by first shifting the two match images, and calculating the squared difference with the reference image. Next, the two disparity images are added to form the difference image. Finally, in the example code, the result image is obtained by performing a convolution with a 13×13 uniform filter and minimizing over results obtained previously.

With our software architecture we have implemented two versions of the algorithm that differ only in the manner in which the pixels in the 13×13 window are summed. The pseudo code of Listing 7.2 shows the version that performs a full 2-dimensional convolution, which we refer to as ***VisSlow***. As explained in detail in [43], a faster sequential implementation is obtained when partial sums in the image's y-direction are buffered while sliding the window over the image. We refer to this optimized version of the algorithm as ***VisFast***.

```
ErrorIm = UnPixOp(ErrorIm, "set", MAXVAL);
FOR all displacements d DO
  DisparityIm1 = BinPixOp(MatchIm1, "horshift", d);
  DisparityIm2 = BinPixOp(MatchIm2, "horshift", 2 × d);
  DisparityIm1 = BinPixOp(DisparityIm1, "sub", ReferenceIm);
  DisparityIm2 = BinPixOp(DisparityIm2, "sub", ReferenceIm);
  DisparityIm1 = BinPixOp(DisparityIm1, "pow", 2);
  DisparityIm2 = BinPixOp(DisparityIm2, "pow", 2);
  DifferenceIm = BinPixOp(DisparityIm1, "add", DisparityIm2);
  DifferenceIm = GenConvOp(DifferenceIm, "mult", "add", unitKer);
  ErrorIm = BinPixOp(ErrorIm, "min", DifferenceIm);
OD
```

Listing 7.2: *Pseudo code for multi-baseline stereo vision.*

7.3.2 Parallel Execution

The generated optimal schedule for either version of the program of Section 7.3.1 requires not more than five communication steps. In the first loop iteration — and only then — the three input images `MatchIm1`, `MatchIm2`, and `ReferenceIm` are scattered to all processing units. The decompositions of these images are all identical (and performed in a row-wise fashion only – i.e., using a $1 \times P$ logical CPU grid mapping) to avoid a domain mismatch and unnecessary communication. Also, in each loop iteration border communication is performed in either version of the program. Again, the extent of the border in each dimension is half the size of the kernel minus one pixel (i.e., six pixels in total). Finally, at the end of the last loop iteration the result image (`ErrorIm`) is gathered to one processor. As in the example of Section 7.2, the optimized parallel programs obtained with our software architecture execute in exactly the same way as would have been the case for reasonable hand-coded implementations.

7.3.3 Performance Evaluation

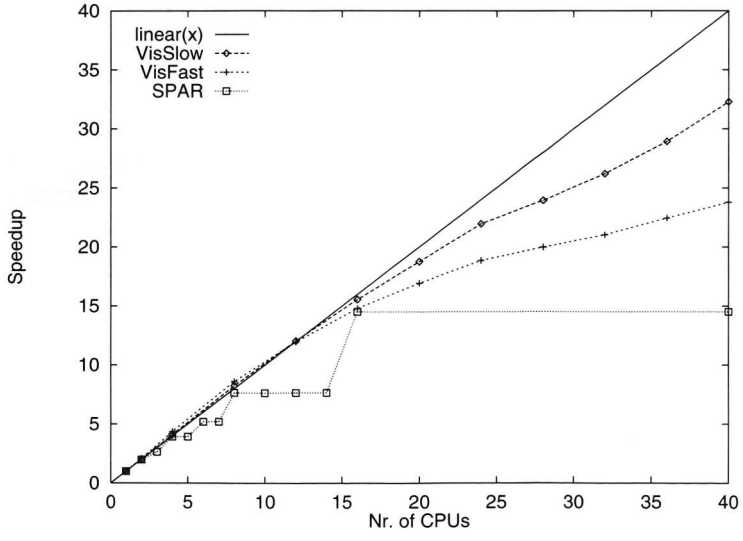
Results obtained for the two implementations, given input images of size 240×256 pixels (as used most often in the literature) are shown in Figure 7.3(a). Given the fact that we only allow border exchange among neighboring nodes in a logical CPU grid, the maximum number of nodes that can be used for such image size is 40. In case more CPUs are being used, several nodes will have partial image structures with an extent of less than 6 pixels in one dimension (due to the one-dimensional partitioning of the input images). As the size of the shadow region for a 13×13 kernel is 6 pixels in both dimensions, nodes would have to obtain data from its neighbor's neighbors as well – or even further away. The communication pattern for this behavior is costly (i.e., the communication versus computation ratio is high), and therefore we have not incorporated it in our architecture.

As expected, Figure 7.3(a) shows that the performance of the ***VisFast*** version of the algorithm is significantly better than that of ***VisSlow***. Also, the graph of Figure 7.3(b) shows that the speedup obtained for both applications is close to linear up to 24 CPUs. When more than 24 nodes are being used, the speedup graphs flatten out due to the relatively short execution times. Because the generated schedule for this program is identical to what an expert programmer would have implemented by hand, this is to be considered optimal. This also can be derived from the fact that superlinear speedups are obtained for up to 12 processing units. Figure 7.4 shows similar speedup characteristics obtained for a system of up to 80 nodes, and using input images of size 512×528 pixels. For up to 40 nodes these results are almost identical to Figure 7.3, indicating a similar impact of communication on overall performance.

In Figures 7.3 and 7.4 we have also made a comparison with results obtained for the same application — implemented in a task parallel manner — written in a specialized parallel programming language (SPAR [129]), and executed on the same parallel machine. In this implementation, referred to as ***VisTask***, each iteration is designated as an independent task, thus exploiting 16 processing units at maximum. For this comparison, the code generated by the SPAR front-end was compiled in a iden-

# CPUs	Software Architecture		SPAR
	VisFast (s)	VisSlow (s)	VisTask (s)
1	1.998	5.554	8.680
2	0.969	2.759	4.372
4	0.458	1.354	2.214
8	0.232	0.674	1.135
12	0.167	0.461	1.135
16	0.135	0.357	0.598
20	0.118	0.296	
24	0.106	0.253	
28	0.100	0.232	
32	0.095	0.212	
36	0.089	0.192	
40	0.084	0.172	

(a)

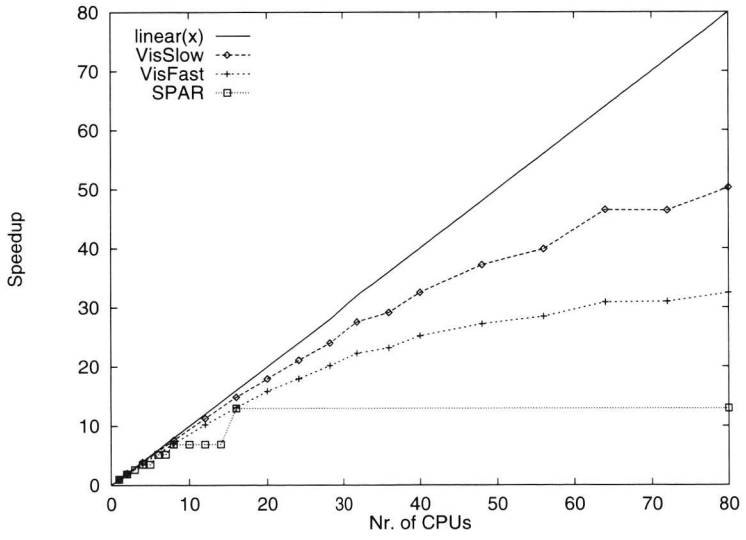


(b)

Figure 7.3: *Performance and speedup characteristics for multi-baseline stereo vision using input images of 240×256 (4-byte) pixels. (a) Execution times in seconds for the optimized parallel programs obtained with our architecture for both algorithms. Results in gray obtained for the task parallel implementation in the SPAR parallel programming language. (b) Speedup graph for all measurements.*

# CPUs	Software Architecture		SPAR
	VisFast (s)	VisSlow (s)	VisTask (s)
1	8.770	24.375	42.993
2	4.515	12.343	22.776
4	2.396	6.300	12.283
8	1.250	3.218	6.219
16	0.670	1.641	3.312
24	0.488	1.156	
32	0.394	0.885	
40	0.348	0.749	
48	0.322	0.655	
56	0.308	0.611	
64	0.284	0.524	
80	0.270	0.485	

(a)



(b)

Figure 7.4: *Performance and speedup characteristics for multi-baseline stereo vision using input images of 512×528 (4-byte) pixels. (a) Execution times in seconds for the optimized parallel programs obtained with our architecture for both algorithms. Results in gray obtained for the task parallel implementation in the SPAR parallel programming language. (b) Speedup graph for all measurements.*

tical manner to the previous case. Although the communication characteristics of the SPAR implementation are significantly different, measurements on a single node indicate that the overhead by our software architecture is much smaller than that of the SPAR runtime system. Nevertheless, the speedup obtained for the ***VisTask*** implementation indicates that SPAR successfully exploits all available parallelism for this particular application. From this comparison we conclude that our software architecture provides fast sequential code, as well as high parallelization efficiency.

Interestingly, our results are comparable to the performance obtained for a ***VisFast***-like implementation in the Adapt parallel image processing language reported by Webb [166] (see Figure 7.5). A comparison is difficult, however, as results were obtained on a significantly different machine (i.e., a collection of iWarp processors, with a better potential for obtaining high speedup than the DAS cluster), and for an implementation optimized for 2^x nodes. Comparison with the speedup characteristics of the Adapt implementation is even more difficult, as the results in Figure 7.6 indicate that they fluctuate substantially. Yet, our results on the DAS (which was installed less than 5 years later) make a strong case for our general purpose approach.

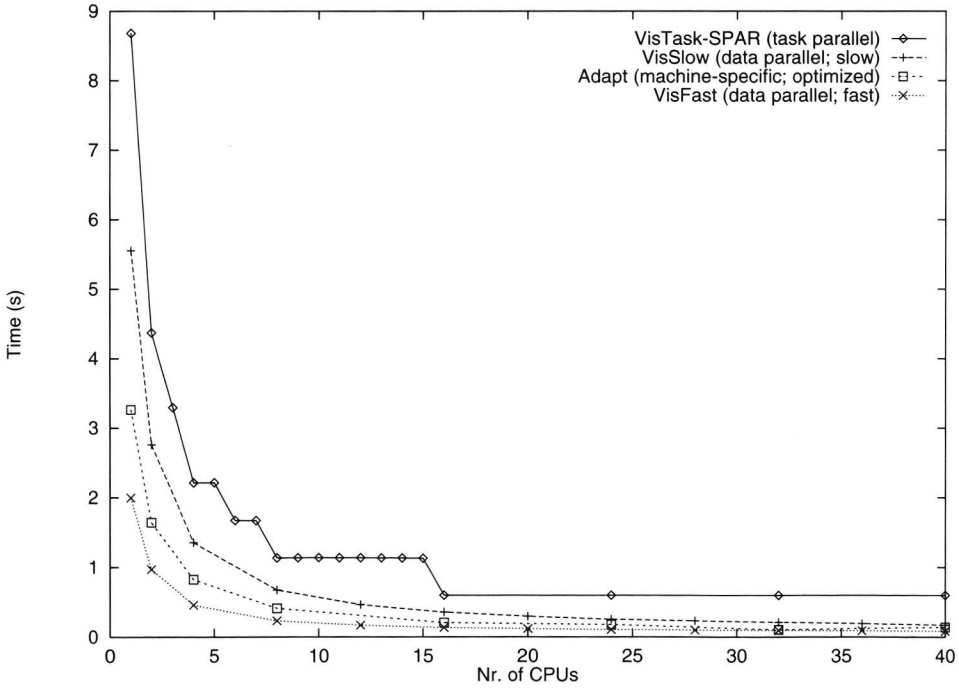


Figure 7.5: *Comparison of execution times for the* ***VisSlow*** *and* ***VisFast*** *programs implemented with our software architecture, the* ***VisTask*** *program implemented using the SPAR parallel language, and the results obtained for the Adapt implementation reported in [166] (all for 240×256 (4-byte) input images).*

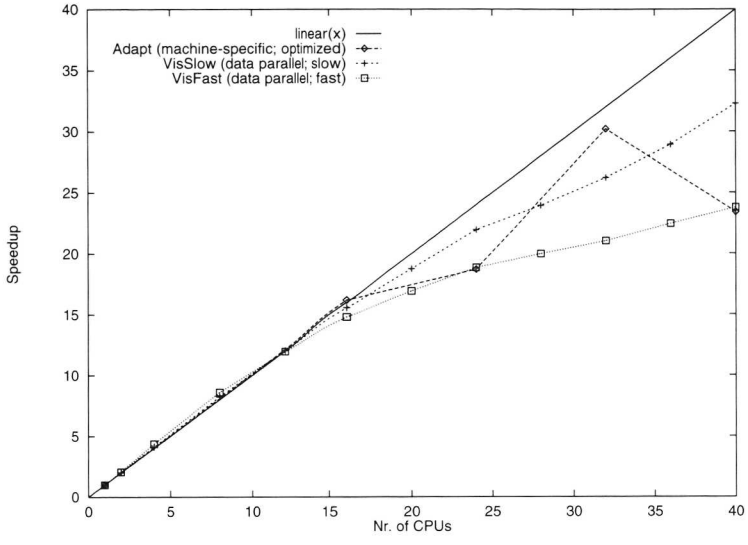


Figure 7.6: *Comparison of speedup for the* ***VisSlow*** *and* ***VisFast*** *programs implemented with our software architecture, and the Adapt implementation reported in [166] (all for 240×256 (4-byte) input images).*

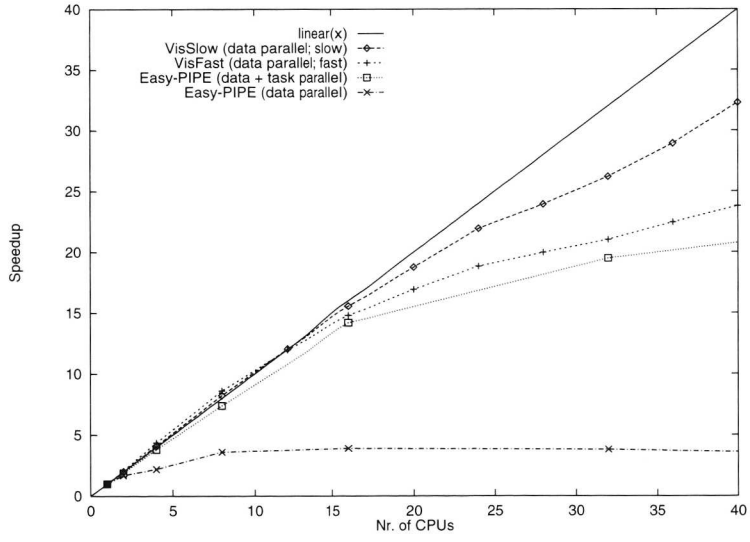


Figure 7.7: *Comparison of speedup for the* ***VisSlow*** *and* ***VisFast*** *programs implemented with our software architecture, and the two Easy-PIPE implementations reported in [111] (all for 240×256 (4-byte) input images).*

More relevant is a comparison with *Easy-PIPE* [111, 112], a library-based software environment for parallel image processing similar to ours. *Easy-PIPE* mainly differs from our architecture in that it incorporates a mechanism for combining data and task parallelism. Also, *Easy-PIPE* does not shield *all* parallelism from the application programmer. As a consequence, *Easy-PIPE* has the potential of outperforming our architecture, which is fully user transparent, and strictly data parallel. Results for the multi-baseline stereo application obtained on the same DAS cluster (see Figure 7.7) indicate that our architecture performs better nonetheless. Part of the difference is accounted to the fact that the two *Easy-PIPE* implementations do not fully exploit all parallelism available in the program. Also, in contrast to our library implementations, the communication routines applied in *Easy-PIPE* rely on the costly creation of separate send and receive buffers in user-space. The bulk of the difference, however, is due to the absence in the *Easy-PIPE* architecture of an inter-operation optimization mechanism for removal of redundant communication overhead, such as our lazy parallelization approach of Chapter 6. As a result, the parallelization overhead of the *Easy-PIPE* implementations is much higher than that of our software architecture.

7.4 Detection of Linear Structures

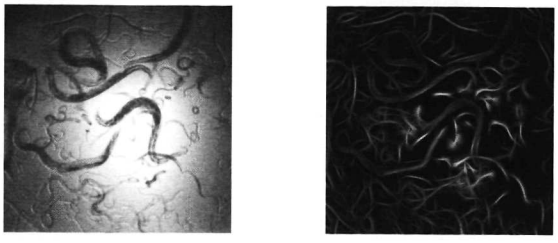
As discussed in [55], the important problem of detecting lines and linear structures in images is solved by considering the second order directional derivative in the gradient direction, for each possible line direction. This is achieved by applying anisotropic Gaussian filters, parameterized by orientation θ, smoothing scale σ_u in the line direction, and differentiation scale σ_v perpendicular to the line, given by

$$r''(x, y, \sigma_u, \sigma_v, \theta) = \sigma_u \sigma_v \left| f_{vv}^{\sigma_u, \sigma_v, \theta} \right| \frac{1}{b^{\sigma_u, \sigma_v, \theta}}, \tag{7.3}$$

with b the line brightness. When the filter is correctly aligned with a line in the image, and σ_u, σ_v are optimally tuned to capture the line, filter response is maximal. Hence, the per pixel maximum line contrast over the filter parameters yields line detection:

$$R(x, y) = \arg \max_{\sigma_u, \sigma_v, \theta} r''(x, y, \sigma_u, \sigma_v, \theta). \tag{7.4}$$

Figure 7.8(a) gives a typical example of an image used as input to this algorithm. Results obtained for a reasonably large subspace of $(\sigma_u, \sigma_v, \theta)$ are shown in Figure 7.8(b).



(a) (b)

Figure 7.8: *Detection of* C. Elegans *worms (Janssen Pharmaceuticals, Belgium).*

```
FOR all orientations θ DO
  RotatedIm = GeometricOp(OriginalIm, "rotate", θ);
  ContrastIm = UnPixOp(ContrastIm, "set", 0);
  FOR all smoothing scales σ_u DO
    FOR all differentiation scales σ_v DO
      FiltIm1 = GenConvOp(RotatedIm, "gaussXY", σ_u, σ_v , 2, 0);
      FiltIm2 = GenConvOp(RotatedIm, "gaussXY", σ_u, σ_v , 0, 0);
      DetectedIm = BinPixOp(FiltIm1, "absdiv", FiltIm2);
      DetectedIm = BinPixOp(DetectedIm, "mul", σ_u × σ_v);
      ContrastIm = BinPixOp(ContrastIm, "max", DetectedIm);
    OD
  OD
  BackRotatedIm = GeometricOp(ContrastIm, "rotate", −θ);
  ResultIm = BinPixOp(ResultIm, "max", BackRotatedIm);
OD
```

Listing 7.3: *Pseudo code for the* ***ConvRot*** *algorithm.*

7.4.1 Sequential Implementations

The anisotropic Gaussian filtering problem can be implemented sequentially in many different ways. In the remainder of this section we will consider three possible approaches. First, for each orientation θ it is possible to create a new filter based on σ_u and σ_v. In effect, this yields a rotation of the filters, while the orientation of the input image remains fixed. Hence, a sequential implementation based on this approach (which we refer to as ***Conv2D***) implies full 2-dimensional convolution for each filter.

The second approach (referred to as ***ConvUV***) is to decompose the anisotropic Gaussian filter along the perpendicular axes u, v, and use bilinear interpolation to approximate the image intensity at the filter coordinates. Although comparable to the ***Conv2D*** approach, ***ConvUV*** is expected to be faster due to a reduced number of accesses to the image pixels. A third possibility (called ***ConvRot***) is to keep the orientation of the filters fixed, and to rotate the input image instead. The filtering now proceeds in a two-stage separable Gaussian, applied along the x- and y-direction.

Pseudo code for the ***ConvRot*** algorithm is given in Listing 7.3. The program starts by rotating the original input image for a given orientation θ. In addition, for all (σ_u, σ_v) combinations the filtering is performed by xy-separable Gaussian filters.

```
FOR all orientations θ DO
  FOR all smoothing scales σ_u DO
    FOR all differentiation scales σ_v DO
      FiltIm1 = GenConvOp(OriginalIm, "func", σ_u, σ_v , 2, 0);
      FiltIm2 = GenConvOp(OriginalIm, "func", σ_u, σ_v , 0, 0);
      ContrastIm = BinPixOp(FiltIm1, "absdiv", FiltIm2);
      ContrastIm = BinPixOp(ContrastIm, "mul", σ_u × σ_v);
      ResultIm = BinPixOp(ResultIm, "max", ContrastIm);
```

Listing 7.4: *Pseudo code for the* ***Conv2D*** *and* ***ConvUV*** *algorithms, with* `"func"` *either* `"gauss2D"` *or* `"gaussUV"`.

For each orientation step the maximum response is combined in a single contrast image structure. Finally, the temporary contrast image is rotated back to match the orientation of the input image, and the maximum response image is obtained.

For the ***Conv2D*** and ***ConvUV*** algorithms, the pseudo code is identical and given in Listing 7.4. Filtering is performed in the inner loop by either a full two-dimensional convolution (***Conv2D***) or by a separable filter in the principle axes directions (***ConvUV***). On a state-of-the-art sequential machine either program may take from a few minutes up to several hours to complete, depending on the size of the input image and the extent of the chosen parameter subspace. Consequently, for the directional filtering problem parallel execution is highly desired.

7.4.2 Parallel Execution

Automatic optimization of the ***ConvRot*** program has resulted in an optimal schedule, as described in more detail Section 4.5.2. In this schedule, the full `OriginalIm` structure is broadcast to all nodes before each calculates its respective partial `RotatedIm` structure. This broadcast needs to be performed only once, as `OriginalIm` is not updated in any operation. Subsequently, all operations in the innermost loop are executed locally on partial image data structures. The only need for communication is in the exchange of image borders in the two Gaussian convolution operations.

The two final operations in the outermost loop are executed in a data parallel manner as well. As this requires the distributed image `ContrastIm` to be available in full at each node (see Section 4.5.2), a gather-to-all operation is performed. Finally, a partial maximum response image `ResultIm` is calculated on each node, which requires a final gather operation to be executed just before termination of the program.

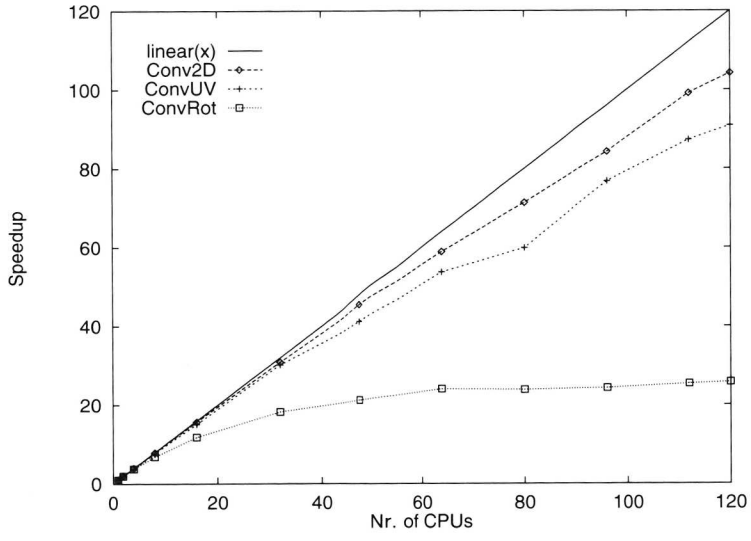
The schedule generated for either the ***Conv2D*** program or the ***ConvUV*** program is straightforward, and similar to that of the template matching application of Section 7.2. First, the `OriginalIm` structure is scattered such that each node obtains an equal-sized non-overlapping slice of the image's domain. Next, all operations are performed in parallel, with a border exchange required in the convolution operations. Finally, before termination of the program `ResultIm` is gathered to a single node.

7.4.3 Performance Evaluation

From the description above it is clear that the ***ConvRot*** algorithm is most difficult to parallelize efficiently. Note that this is due to the data dependencies present in the algorithm (i.e., the repeated image rotations), and not in any way related to the capabilities of our software architecture. In other words, even when implemented by hand the ***ConvRot*** algorithm is expected to have speedup characteristics that are not as good as those of the other two algorithms. Furthermore, ***Conv2D*** is expected to be the slowest sequential implementation, due to the excessive accessing of image pixels in the 2-dimensional convolution operations. In general, ***ConvUV*** and ***ConvRot*** will be competing for the best sequential performance, depending on the amount of filtering performed for each orientation.

# CPUs	ConvRot (s)	Conv2D (s)	ConvUV (s)
1	666.720	2085.985	437.641
2	337.877	1046.115	220.532
4	176.194	525.856	113.526
8	97.162	264.051	56.774
16	56.320	132.872	28.966
32	36.497	67.524	14.494
48	31.399	45.849	10.631
64	27.745	35.415	8.147
80	27.950	29.234	7.310
96	27.449	24.741	5.697
112	26.284	21.046	5.014
120	25.837	20.017	4.813

(a)



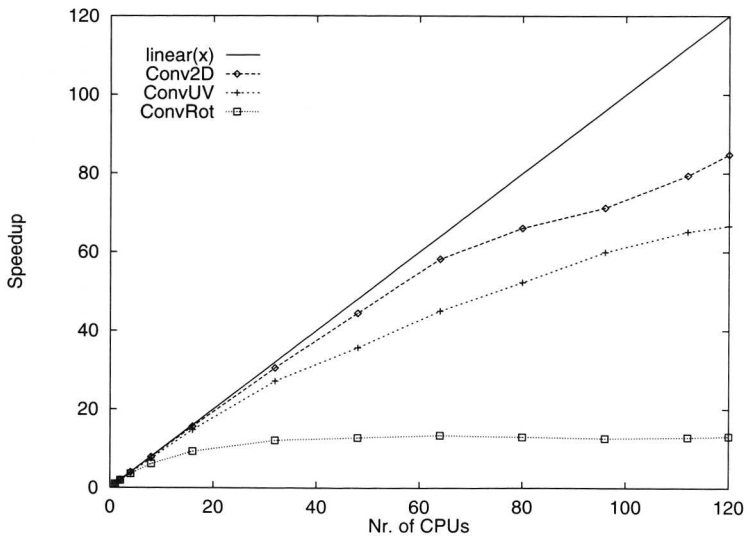
(b)

Figure 7.9: *(a) Performance and (b) speedup characteristics for computing a typical orientation scale-space at 5° angular resolution (i.e., 36 orientations) and 8 (σ_u, σ_v) combinations. Scales computed are $\sigma_u \in \{3,5,7\}$ and $\sigma_v \in \{1,2,3\}$, ignoring the isotropic case $\sigma_{u,v} = \{3,3\}$. Image size is 512×512 (4-byte) pixels.*

Figure 7.9 shows that these expectations are indeed correct. On one processor ***ConvUV*** is about 1.5 times faster than ***ConvRot***, and about 4.8 times faster than ***Conv2D***. For 120 nodes these factors have become 5.4 and 4.1 respectively. Because of the relatively poor speedup characteristics, ***ConvRot*** even becomes slower than ***Conv2D*** when the number of nodes becomes large. Although ***Conv2D*** has better speedup characteristics, the ***ConvUV*** implementation always is fastest, either sequentially or in parallel. Figure 7.10 presents similar results for a minimal parameter subspace, thus indicating a lower bound on the obtainable speedup.

# CPUs	ConvRot (s)	Conv2D (s)	ConvUV (s)
1	110.127	325.259	56.229
2	56.993	162.913	28.512
4	30.783	82.092	14.623
8	17.969	41.318	7.510
16	11.874	20.842	3.809
32	9.102	10.660	2.071
48	8.617	7.323	1.578
64	8.222	5.589	1.250
80	8.487	4.922	1.076
96	8.729	4.567	0.938
112	8.551	4.096	0.863
120	8.391	3.836	0.844

(a)



(b)

Figure 7.10: *(a) Performance and (b) speedup characteristics for computing a minimal orientation scale-space at* 15° *angular resolution (i.e., 12 orientations) and 2* (σ_u, σ_v) *combinations. Scales computed are* $\sigma_{u,v} = \{1, 3\}$ *and* $\sigma_{u,v} = \{3, 7\}$.

The generated schedules for both the ***Conv2D*** program and the ***ConvUV*** program are identical to what an expert would have implemented by hand. Speedup values obtained on 120 nodes for a typical parameter subspace (Figure 7.9) are 104.2 and 90.9 for ***Conv2D*** and ***ConvUV*** respectively. As a result we can conclude that our software architecture behaves well for these implementations. In contrast, the usage of algorithmic patterns (see Chapter 3) has caused the sequential implementation of image rotation to be non-optimal for certain special cases. As an example, rotation over 90° can be implemented much more efficiently than rotation over any arbitrary angle. In our architecture we have decided not to do so, mainly for reasons

of software maintainability (see Chapter 2). As a result, we expect a hand-coded and hand-optimized version of the same algorithm to be faster, but only marginally so.

7.5 Conclusions and Future Work

In this chapter we have given an assessment of the effectiveness of our software architecture in providing significant performance gains. To this end, we have described the sequential implementation, as well as the automatic parallelization, of three different example applications. The applications are relevant for this evaluation, as all are well-known from the literature, and all contain many fundamental operations required in many other image processing research areas as well.

The results presented in this chapter have shown our software architecture to serve well in obtaining highly efficient parallel applications. Moreover, in almost all situations handcrafted code would not have produced significantly better results. As such, we have shown that our architecture adheres to requirement I.2 put forward in Section 2.3 - - which states that the obtained efficiency generally should compare well to that of reasonable hand-coded parallel implementations. As indicated in Section 7.4.3, however, for certain specific operations we have decided that code maintainability is more important than highest performance. Consequently, in comparison with optimal handcrafted parallel code, any application that makes extensive use of such operations may suffer from reduced efficiency (but often only marginally so).

As an important note we should state that, although all parallelism is hidden inside the architecture itself, much of the efficiency of parallel execution is still in the hands of the application programmer. As we have shown in Section 7.4.3, if a sequential implementation is provided that requires costly communication steps when executed in parallel, program efficiency may be disappointing. Thus, for highest performance the application programmer still should be aware of the fact that usage of such operations is expensive, and should be avoided whenever possible. Any programmer knows that this requirement is not new, however, as a similar requirement holds for sequential execution as well. In other words, this is not a drawback that results from any of the design choices incorporated in our software architecture. The problem can not be avoided, as it stems directly from the fact that all parallelization and optimization issues are shielded from the application programmer entirely.

In conclusion: although we are aware of the fact that a much more extensive evaluation is required to obtain more insight in the specific strengths and weaknesses of our architecture, the presented results clearly indicate that our architecture constitutes a powerful and user-friendly tool for obtaining high performance in many important image processing research areas. For future evaluation, we will continue implementing example applications to investigate the implication of parallelization of typical image processing problems, especially in the area of real-time image processing.

Chapter 8

Summary and Discussion

"From a word to a word I was led to a word,
From a deed to another deed."

Excerpt from *The Poetic Edda* (Iceland, ca. 1280)

8.1 Summary

To satisfy the performance requirements of current and future applications in image-, video-, and multimedia processing, the image processing community at large exhibits an overwhelming desire to employ the speed potential of high performance computer architectures. Unfortunately, there is a major discrepancy between the need for *easily obtainable* speed in imaging, and the potential of current high performance computers to fulfill this need. Primarily, we ascribe this problem to the fact that no programming tool is available that serves as an effective aid in the development of image processing applications for parallel and distributed systems. Existing tools generally require the user to have a thorough insight in the complexities of parallelization, often at a level of detail far beyond that of non-dedicated parallel programmers. As it is unrealistic to expect researchers in imaging to become experts in high performance computing, it is essential to provide a tool that shields its users from all intricacies of parallelization.

The work described in this thesis is an endeavor to bridge the gap between the expertise of researchers in image processing, and the particular skills required for efficient employment of high performance hardware architectures. To this end, we describe the design and implementation of an innovative software architecture that allows its users to develop parallel image processing applications in a *user transparent* (i.e., fully sequential) manner. We explore the requirements such architecture must adhere to for it to serve as a long-term solution for the image processing community. Also, we provide a detailed discussion of each of the architecture's constituent components, and the research issues associated with each of these. Finally, we evaluate the provided performance gains, to see how these compare to reasonable hand-coded applications.

In Chapter 2, we investigate the applicability of existing high performance hardware architectures and associated parallelization tools in the field of image processing. Based on a set of requirements we conclude that homogeneous Beowulf-type commodity clusters constitute the most appropriate class of target platforms — most importantly due to the emphasis on price-performance. The evaluation of associated software tools shows library-based environments to offer a solution that is most likely to be acceptable to the image processing community. Primarily, this is because these are most easily provided with a programming model that offers full *user transparency*. However, due to insufficient *sustainability* levels, no user transparent tool is found to provide an acceptable *long-term* solution. Based on these observations we propose a new library-based software architecture for parallel image processing on homogeneous Beowulf-type commodity clusters. Due to its innovative design and implementation the architecture fully adheres to the requirements of user transparency and long term sustainability. Consequently, the architecture constitutes a solution that is likely to be acceptable as a long-term solution for the image processing community at large.

In Chapter 3, we present the design philosophy behind the parallel image processing library, which is the core component of the developed software architecture. Primarily, we focus on the problem of implementing the library such that code redundancy is avoided as much as possible, whilst ensuring efficiency of parallel execution. To this end, we introduce the notion of *parallelizable patterns*, and discuss how parallel implementations are easily obtained by sequential concatenation of operations that are *separately available* in the library. More specifically, on the basis of a set of four *data access pattern types*, we define a default parallelization strategy for any operation that maps onto one of two parallelizable pattern types. For each parallel operation this default strategy is optimal, as it fully exploits the available parallelism with minimal communication overhead. As such, we demonstrate that the presented design philosophy allows for long-term architecture sustainability, as well as high efficiency.

In Chapter 4, we indicate how to apply a simple analytical performance model in the process of automatic parallelization and optimization of complete image processing applications. Existing approaches generally incorporate a direct relationship between the estimation accuracy and the model's complexity (and thus: efficiency of evaluation). To deal with this problem, we propose a *semi-empirical modeling* technique. While being simple and portable, the approach also provides a sufficiently high estimation accuracy. The approach is based on a high level abstract parallel image processing machine (or *APIPM*) definition, which is designed to capture typical run time behavior of parallel low level image operations. From the related APIPM instruction set, a high level model is obtained that is applicable to all machines in the class of target platforms. The essence of the semi-empirical modeling approach is that any behavior or cost factor that can not be assumed identical for all target platforms (such as interprocess communication, or caching) is abstracted from in the definition of the model parameters. To still bind each abstract model parameter to an accurate performance estimation for a parallel machine at hand, benchmarking is performed on a small set of sample data to capture all such essential, but implicit cost factors. A comparison of model estimations and experimental measurements indicates that, for realistic applications, the APIPM-based performance models are highly accurate.

Chapter 5 extends the APIPM-based performance models for more accurate estimation of the MPI message passing primitives used in the library implementations. Existing communication models (such as LogP) do not incorporate all capabilities of MPI's send and receive operations. So far, the (often significant) effect of memory layout on communication costs has been ignored completely. In our software architecture, a higher predictive power is essential to perform the important task of automatic and optimal distribution of image data structures. To this end, we define a new model (called *P-3PC*), that closely matches the behavior of MPI's standard point-to-point operations. First, the model accounts for differences in performance at the sender, the receiver, and the full communication path. Also, it models the impact of memory layout, and accounts for communication costs that are not linearly dependent on message size. Experiments performed on two significantly different cluster architectures indicate that, in comparison with related models, P-3PC is capable of more accurate estimation of the communication overhead of typical image processing applications.

Chapter 6 discusses the automatic conversion of any sequential image processing application into a legal, correct, and efficient parallel version. To this end, we define a finite state machine (fsm) specification that guarantees the process to be performed correctly at all times. First, the fsm is shown to bring about a surprisingly simple and efficient approach (called *lazy parallelization*) for communication cost minimization. For further optimization, the fsm is used in the construction of an application state transition graph (*ASTG*), that characterizes an application's run time behavior, and also incorporates all possible (combinations of) parallelization and optimization decisions. As each decision is annotated with a run time cost estimation obtained from the APIPM-based performance models, the fastest version of the program is represented by the cheapest branch in the ASTG. As the issue of automatic optimization of complete applications is the central, most essential problem our software architecture for user transparent parallel image processing is confronted with, the applied solution combines all of the results obtained in Chapters 3, 4, and 5.

In Chapter 7, we give an assessment of our architecture's effectiveness in providing significant performance gains. We describe the implementation and automatic parallelization of three well-known example applications that contain many operations commonly applied in image processing research. From the evaluation, we conclude that the performance obtained with the parallel versions generated by our software architecture compares well to that of reasonable hand-coded parallel implementations.

8.2 Discussion

In this thesis we have aimed at the development of an effective programming tool that provides sustainable support in the implementation of parallel image processing software by non-experts in high-performance computing. We believe that we have succeeded in that mission. In the very first place because the architecture shields the user from *all* parallelization and optimization issues. As such, the architecture can be used immediately, without requiring additional knowledge from the application programmer. In the second place because the architecture allows its developers to

respond to changing demands and environments quickly and elegantly. The applied design philosophy has largely inherited this property from the sequential image library (Horus) the architecture is based on (see Chapter 3). Finally, because the obtained efficiency was shown to be comparable to that of reasonable hand-optimized code. This implies that the parallelization overhead induced by the architecture is marginal. For these reasons we conclude that the software architecture fully adheres to the requirements of user transparency and sustainability as put forward in Section 2.3.1.

Despite this result, certain properties of the software architecture as described in this thesis are not always desirable. A first problem is due to the extensive use of abstractions incorporated in the architecture. As pointed out in Chapter 3, among the advantages of the use of abstraction and (parallelizable) patterns are a huge reduction in human software engineering effort, and enhanced software maintainability, extensibility, reusability, and portability. However, there is a trade-off between the use of specific and abstract libraries in terms of (sequential) processing speed. Also, abstract libraries may have a long compilation time and large footprints due to the automatic expansion of function instantiations. One way to overcome these disadvantages is to build a tool that can automatically generate specific (even tailor-made) image processing libraries from the existing abstract implementations.

A second issue that was not discussed in this thesis, is the question of how to deal with enormous amounts of input data. Applications working on video-sequences, or complete image databases, may suffer from a significant I/O performance bottleneck. Therefore, it is essential to re-evaluate data I/O in our architecture, and to incorporate optimizations accordingly. One solution may be to use data compression techniques in software. However, research related to this issue may also lead to the conclusion that hardware extensions (e.g., MPEG-encoders and -decoders) are essential.

A problem with *any* library-based environment is that it can never provide a *complete coverage* of all desired functionality. As stated in Chapter 3, we estimate that the algorithmic patterns available in our library cover over 90% of all low level imaging operations. Additional patterns, such as for recursive neighborhood operations, and queue-based algorithms are currently under construction. Other algorithms (a.o., data dependent operation) may not map onto one of the standard patterns, and also may not parallelize well, because of irregular data access. In spite of this, we do not expect a large amount of patterns to be added still, as one can compute only a limited variety.

In conclusion: the work described in this thesis indicates that it is possible to provide an effective long-term solution to a difficult problem, basically by incorporating a set of relevant high-level abstractions, and applying relatively simple methods for resolution of each constituent sub-problem. We believe that a similar approach could be applicable in other fields of research as well, especially in areas where the set of typical operations is limited — as is the case in low level image processing. However, only time will tell whether our software architecture for user transparent parallel image processing indeed is the effective long-term solution we consider it to be.

Bibliography

[1] A. Alexandrov, M. Ionescu, K. Schauser, and C. Scheiman. LogGP: Incorporating Long Messages into the LogP Model - One Step Closer Towards a Realistic Model for Parallel Computation. In *Proceedings of the Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 95–105, Santa Barbara, California, USA, July 1995.

[2] R. Allen and S. Johnson. Compiling C for Vectorization, Parallelization, and Inline Expansion. In *Proceedings of the SIGPLAN '88 Conference of Programming Languages Design and Implementation*, pages 241–249, Atlanta, Georgia, USA, June 1988.

[3] C. Andras Moritz and M.I. Frank. LoGPC: Modeling Network Contention in Message-Passing Programs. *IEEE Transactions on Parallel and Distributed Systems*, 12(4):404–415, April 2001.

[4] I. Andreadis, A. Gasteratos, and P. Tsalides. An ASIC for Fast Grey-Scale Dilation. *Microprocessors and Microsystems*, 20(2):89–95, 1996.

[5] Applied Parallel Research. FORGE Explorer User's Guide. Technical report, Version 2.0, 1995.

[6] H.E. Bal. Interprocess Communication and Synchronization based on Message Passing. Technical report, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, The Netherlands, 1995.

[7] H.E. Bal et al. The Distributed ASCI Supercomputer Project. *Operating Systems Review*, 34(4):76–96, October 2000.

[8] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum. Orca: A Language for Parallel Programming of Distributed Systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992.

[9] R. van Balen, D. Koelma, T. ten Kate, B. Mosterd, and A.W.M. Smeulders. SciIImage: A Multi-layered Environment for Use and Development of Image Processing Software. In *Experimental Environments for Computer Vision and Image Processing*, pages 107–126, 1994.

[10] U. Banerjee, R. Eigenmann, A. Nicolau, and D.A. Padua. Automatic Program Parallelization. *Proceedings of the IEEE*, 81(2):211–243, February 1993.

[11] A. Bar-Noy and S. Kipnis. Designing Broadcasting Algorithms in the Postal Model for Message-Passing Systems. *Mathematical Systems Theory*, 27(5):431–452, 1994.

[12] G. Barnes et al. The ILLIAC IV Computer. *IEEE Transactions on Computers*, C-17(8):746–757, August 1968.

[13] J. Barnes. *Programming in Ada 95, first edition.* Addison Wesley, 1995.

[14] G. Baumgartner et al. A Performance Optimization Framework for Compilation of Tensor Contraction Expressions into Parallel Programs. In *Proceedings of the 16th International Parallel & Distributed Processing Symposium - Workshop on High-Level Parallel Programming Models and Supportive Environments*, Fort Lauderdale, Florida, USA, April 2002.

[15] B.N. Bershad, M.J. Zekauskas, and W. A. Sawdon. The Midway Distributed Shared Memory System. In *Proceedings of the 38th IEEE International Computer Conference*, pages 528–537, San Francisco, California, USA, February 1993.

[16] R.A.F. Bhoedjang, T. Rühl, and H.E. Bal. LFC: A Communication Substrate for Myrinet. In *Proceedings of the Fourth Annual Conference of the Advanced School for Computing and Imaging*, pages 31–37, Lommel, Belgium, June 1998.

[17] W. Blume et al. Automatic Detection of Parallelism: A Grand Challenge for High-Performance Computing. *IEEE Parallel and Distributed Technology*, 2(3):37–47, 1994.

[18] A. Bouridane et al. A High Level FPGA-based Abstract Machine for Image Processing. *Journal of Systems Architecture*, 45(10):809–824, April 1999.

[19] S. Boussakta. A Novel Method for Parallel Image Processing Applications. *Journal of Systems Architecture*, 45:825–839, 1999.

[20] J. Brown and D. Crookes. A High Level Language for Parallel Image Processing. *Image and Vision Computing*, 12(2):67–79, March 1994.

[21] J. Bruck, L. de Coster, N. Dewulf, C.-T. Ho, and R. Lauwereins. On the Design and Implementation of Broadcast and Global Combine Operations Using the Postal Model. *IEEE Transactions on Parallel and Distributed Systems*, 7(3):256–265, March 1996.

[22] D.K.G. Campbell. A Survey of Models of Parallel Computation. Technical Report YCS-97-278, Department of Computer Science, University of York, March 1997.

[23] B.L. Chamberlain et al. A Comparative Study of the NAS MG Benchmark across Parallel Languages and Architectures. In *Proceedings of the 2000 ACM/IEEE Supercomputing Conference on High Performance Networking and Computing*, Dallas, Texas, USA, November 2000.

[24] S.C. Chan, H.O. Ngai, and K.L. Ho. A Programmable Image Processing System using FPGAs. *International Journal of Electronics*, 75(4):725–730, 1993.

[25] R. Chandra, L. Dagum, D. Kohr, and D. Maydan. *Parallel Programming in OpenMP*. Morgan Kaufmann, October 2000.

[26] K.M. Chandy and C. Kesselman. Compositional C++: Compositional Parallel Programming. Technical Report TR-92-13, California Institute of Technology, 1992.

[27] S. Chatterjee et al. Generating Local Addresses and Communication Sets for Data Parallel Programs. *Journal of Parallel and Distributed Computing*, 26(1):72–84, 1995.

[28] D. Chavarría-Miranda et al. Data-Parallel Compiler Support for Multipartitioning. In *Proceedings of the 7th International Euro-Par Conference (Euro-Par 2001), LNCS 2150*, pages 241–253, Manchester, UK, August 2001.

[29] M. Chu-Carroll and L.L. Pollock. Design and Implementation of a General Purpose Parallel Programming System. In *Proceedings of HPCN Europe 1996*, pages 499–507, Brussels, Belgium, April 1996.

[30] Clusters@TOP500. `URL: http://clusters.top500.org/`.

[31] J.M. Constantin, M.W. Berry, and B.T. Vander Zanden. Parallelization of the Hoshen-Kopelman Algorithm Using a Finite State Machine. *International Journal of Supercomputer Applications and High Performance Computing*, 11(1):31–45, 1997.

[32] M. Crochemore and W. Rytter. Note on Two-Dimensional Pattern Matching by Optimal Parallel Algorithms. In *Proceedings of the Second International Conference on Parallel Image Analysis, ICPIA '92*, pages 100–112, Ube, Japan, December 1992.

[33] D. Crookes. Architectures for High Performance Image Processing: The Future. *Journal of Systems Architecture*, 45:739–748, 1999.

[34] D. Crookes et al. Achieving Portability and Efficiency through Automatic Optimisation: an Investigation in Parallel Image Processing. In *Euro-Par 1998*, pages 102–112, Southhampton, UK, September 1998.

[35] D. Crookes, A.P. McHale, and N. Beney. A DAP-based Implementation of a Portable Parallel Image Processing Machine. In *Parallel Processing: CONPAR 92-VAPP V*, pages 803–804, Lyon, France, September 1992.

[36] D. Crookes and P.J. Morrow. Design Considerations for a Portable Parallel Abstract Machine for Low Level Image Processing. In *BCS Workshop on Abstract Machine Models for Highly Parallel Computers*, pages 107–110, Leeds, UK, March 1991.

[37] D. Crookes, P.J. Morrow, and P.J. McParland. IAL: A Parallel Image Processing Programming Language. *IEE Proceedings, Part I*, 137(3):176–182, June 1990.

[38] D. Culler et al. LogP: Towards a Realistic Model of Parallel Computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, San Diego, California, USA, May 1993.

[39] H.J. Curnow and B.A. Wichmann. A Synthetic Benchmark. *The Computer Journal*, 19(1):43–49, February 1976.

[40] A. Darte, D. Chavarría-Miranda, R. Fowler, and J. Mellor-Crummey. Generalized Multipartitioning for Multi-dimensional Arrays. In *Proceedings of the 16th International Parallel & Distributed Processing Symposium*, Fort Lauderdale, Florida, USA, April 2002.

[41] D. Dent, G. Mozdzynski, D. Salmond, and B. Carruthers. Implementation and Performance of OpenMP in ECMWF's IFS Code. In *Proceedings of the Fifth European SGI/Cray MPP Workshop*, Bologna, Italy, September 1999.

[42] K. Diefendorff, P.K. Dubey, R. Hochsprung, and H. Scales. AltiVec Extension to PowerPC Accelerates Media Processing. *IEEE Micro*, 20(2):85–95, March/April 2000.

[43] P. Dinda et al. The CMU Task Parallel Program Suite. Technical Report CMU-CS-94-131, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1994.

[44] J. Dongarra, J.L. Martin, and J. Worlton. Computer Benchmarking: Paths and Pitfalls. *IEEE Spectrum*, 24(7):38–43, July 1987.

[45] M.C. d'Ornellas. *Algorithmic Patterns for Morphological Image Processing*. PhD thesis, Faculty of Science, University of Amsterdam, The Netherlands, March 2001.

[46] B.A. Draper, J.R. Beveridge, A.P.W. Böhm, C. Ross, and M. Chawathe. Implementing Image Applications on FPGAs. In *Proceedings of the 16th International Conference on Pattern Recognition*, pages 1381–1384, Quebec City, Canada, August 2002.

[47] J.T.J. van Eijndhoven, F.W. Sijstermans, K.A. Vissers, E.J.D. Pol, M.J.A. Tromp, P. Struik, R.H.J. Bloks, P. van der Wolf, A.D. Pimentel, and H.P.E. Vranken. TriMedia CPU64 Architecture. In *Proceedings of the IEEE International Conference on Computer Design (ICCD'99)*, pages 586–592, Austin, Texas, USA, October 1999.

[48] J. Eyre and J. Bier. The Evolution of DSP Processors: From Early Architecture to the Latest Developments. *IEEE Signal Processing Magazine*, 17(2):44–55, March 2000.

[49] M.J. Flynn. Very High Speed Computing Systems. *Proceedings of the IEEE*, 56(12):1901–1909, 1966.

[50] I.T. Foster. *Designing and Building Parallel Programs : Concepts and Tools for Parallel Software Engineering.* Addison-Wesley, 1995.

[51] M. Frigo and S.G. Johnson. FFTW: An Adaptive Software Architecture for the FFT. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, pages 1381–1384, Seattle, Washington, USA, May 1998.

[52] E. Gabber, A. Averbuch, and A. Yehudai. Portable, Parallelizing Pascal Compiler. *IEEE Software*, 10(2):71–81, March 1993.

[53] A. Geist et al. *PVM: Parallel Virtual Machine - A Users' Guide and Tutorial for Networked Parallel Computing.* Scientific and Engineering Computation Series. The MIT Press, 1994.

[54] G.A. Geist, J.A. Kohl, and P.M. Papadopoulos. PVM and MPI: a Comparison of Features. *Calculateurs Paralleles*, 8(2), 1996.

[55] J.M. Geusebroek, A.W.M. Smeulders, and H. Geerts. A Minimum Cost Approach for Segmenting Networks of Lines. *Int. Journal of Computer Vision*, 43(2):99–111, July 2001.

[56] J.M. Geusebroek, A.W.M. Smeulders, and J. van de Weijer. Fast Anisotropic Gauss Filtering. In *Proceedings of the 7th European Conference on Computer Vision, Lecture Notes in Computer Science 2350*, pages 99–112, 2002.

[57] R.J. Goozée and P.A. Jacobs. Distributed and Shared Memory Parallelism with a Smoothed Particle Hydrodynamics Code. In *Proceedings of the 10th Biennial Computational Techniques and Applications Conference*, Brisbane, Australia, July 2001.

[58] S. Gregory. *Parallel Logic Programming in PARLOG: The Language and Its Implementation.* Addison Wesley, 1987.

[59] C. Grelck. Array Padding in the Functional Language SAC. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 2553–2560, Las Vegas, Nevada, USA, June 2000.

[60] A.S. Grimshaw. Easy-to-Use Object-Oriented Parallel Processing with Mentat. *IEEE Computer*, 26(5):39–51, May 1993.

[61] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, September 1996.

[62] S.E. Hambrusch. Models for Parallel Computation. In *Proceedings of the 1996 Workshop on Challenges for Parallel Processing*, pages 92–95, August 1996.

[63] S.E. Hambrusch and A. Khokhar. C^3: A Parallel Model for Coarse-Grained Machines. *Journal of Parallel and Distruted Computing*, 32(2):139–154, 1996.

[64] L.G.C. Hamey, J.A. Webb, and I.C. Wu. An Architecture Independent Programming Language for Low Level Vision. *Computer Vision, Graphics and Image Processing*, 48(2):246–264, 1989.

[65] D.W. Hammerstrom and D.P. Lulich. Image Processing Using One-Dimensional Processor Arrays. *Proceedings of IEEE*, 84(7):1005–1018, 1996.

[66] S. Hauck. The Roles of FPGAs in Reprogrammable Systems. *Proceedings of the IEEE*, 86(4):615–639, April 1998.

[67] R. Hempel. The Status of the MPI Message-Passing Standard and Its Relation to PVM. In *Parallel Virtual Machine - EuroPVM'96, Third European PVM Conference*, pages 14–21, Munich, Germany, 1996.

[68] T. Hey. Performance Engineering and the Grid. Invited Talk. Presented at *the 7th International Euro-Par Conference (Euro-Par 2001)*, Manchester, UK, August 2001.

[69] R. Hockney and M. Berry. Public International Benchmarks for Parallel Computers. Technical report, PARKBENCH Committee: Report-1, February 1994. Available at `http://www.netlib.org/parkbench/`.

[70] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation (2nd Edition)*. Addison Wesley, 2000.

[71] D. Howe. FOLDOC - Free On-Line Dictionary Of Computing, March 2001. Available at foldoc.doc.ic.ac.uk.

[72] R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley & Sons, Inc., 1991.

[73] L.H. Jamieson, E.J. Delp, S.E. Hambrusch, A.A. Khokhar, G.W. Cook, F. Hameed, J.N. Patel, and K. Shen. Parallel Scalable Libraries and Algorithms for Computer Vision. In *Proceedings of the 12th International Conference on Pattern Recognition*, volume III, pages 223–228, 1994.

[74] L.H. Jamieson, E.J. Delp, and A.A. Khokhar. A Library-Based Program Development Environment for Parallel Image Processing. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 187–194, Mississippi State, Mississippi, USA, October 1993.

[75] L.H. Jamieson, E.J. Delp, C.-C. Wang, J. Li, and F.J. Weil. A Software Environment for Parallel Computer Vision. *IEEE Computer*, 25(2):73–75, February 1992.

[76] K.L. Johnson, M.F. Kaashoek, and D.A. Wallach. CRL: High-Performance All-Software Distributed Shared Memory. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*. pages 213–226, Copper Mountain, Colorado, USA, December 1995.

[77] G. Jones and M. Goldsmith. *Programming in Occam 2*. International Series in Computer Science. Prentice Hall, 1988.

[78] P.P. Jonker. Why Linear Arrays are Better Image Processors. In *Proceedings of the 12th International Conference on Pattern Recognition*, pages 334–338, Jerusalem, Israel, October 1994.

[79] Z. Juhasz. An Analytical Method for Predicting the Performance of Parallel Image Processing Operations. *The Journal of Supercomputing*, 12(1/2):157–174, 1998.

[80] Z. Juhasz and D. Crookes. A PVM Implementation of a Portable Parallel Image Processing Library. In *Parallel Virtual Machine - EuroPVM'96, Third European PVM Conference*, pages 188–196, Munich, Germany, 1996.

[81] Z. Juhasz, D. Crookes, and A. Chaudry. A Portable, Parallel Image Processing System in Java. In *Proceedings of DAPSYS, Workshop on Parallel and Distributed Systems*, pages 151–154, Budapest, Hungary, September 1998.

[82] T. Kanade. Development of a Video-Rate Stereo Machine. In *Proceedings of the 1994 DARPA Image Understanding Workshop*, pages 549–558, November 1994.

[83] D. Koelma. *A Software Environment for Image Interpretation*. PhD thesis, Faculty of Mathematics, Computer Science, Physics and Astronomy, University of Amsterdam, The Netherlands, March 1996.

[84] D. Koelma et al. Horus C++ Reference, Version 1.1. Technical report, Intelligent Sensory Information Systems, Faculty of Science, University of Amsterdam, The Netherlands, January 2002.

[85] D. Koelma et al. Horus User Guide, Version 1.1. Technical report, Intelligent Sensory Information Systems, Faculty of Science, University of Amsterdam, The Netherlands, January 2002.

[86] D. Koelma, P.P. Jonker, and H.J. Sips. A Software Architecture for Application Driven High Performance Image Processing. In *Parallel and Distributed Methods for Image Processing, Proceedings of SPIE*, volume 3166, pages 340–351, 1997.

[87] D. Koelma and H.J. Sips. A Software Architecture for Parallel Image Processing. In *Proceedings of the Third Annual Conference of the Advanced School for Computing and Imaging*, pages 34–40, Heijen, The Netherlands, June 1997.

[88] E.R. Komen. *Low-Level Image Processing Architectures*. PhD thesis, Delft University of Technology, The Netherlands, 1990.

[89] S. Kyo, T. Koga, and S. Okazaki. IMAP-CE: A 51.2 Gops Video Rate Image Processor with 128 VLIW Processing Elements. In *Proceedings of the 2001 International Conference on Image Processing*, Thessaloniki, Greece, October 2001.

[90] J. Landrum, J. Hardwick, and Q.F. Stout. Predicting Algorithm Performance. *Computing Science and Statistics*, 30:309–314, 1998.

[91] P. Lapsley, J. Bier, A. Shoham, and E.A. Lee. *DSP Processor Fundamentals: Architectures and Features.* IEEE Press Series on Signal Processing, 1996.

[92] M. Lauria. LogP Characterization of FM on the VU's DAS Machine. Technical report, Dipartimento di Informatica e Sistemistica, Universita di Napoli Federico II, 1997.

[93] C. Lee and M. Hamdi. Parallel Image Processing Applications on a Network of Workstations. *Parallel Computing*, 21(1):137–160, January 1995.

[94] C. Lee, Y.-F. Wang, and T. Yang. Static Global Scheduling for Optimal Computer Vision and Image Processing Operations on Distributed-Memory Multiprocessors. In *Computer Analysis of Images and Patterns, 6th International Conference, CAIP '95*, pages 920–925, 1995.

[95] C. Lee, Y.-F. Wang, and T. Yang. Global Optimization for Mapping Parallel Image Processing Tasks on Distributed Memory Machines. *Journal of Parallel and Distributed Computing*, 45(1):29–45, 1997.

[96] P. Linz. *An Introduction to Formal Languages and Automata.* D.C. Heath and Company, 1990.

[97] D.B. Loveman. High Performance Fortran. *IEEE Parallel & Distributed Technology*, pages 25–42, February 1993.

[98] L. Luck and C. Chakrabaty. A Digit-Serial Architecture For Gray-Scale Morphological Filtering. *IEEE Transactions on Image Processing*, 4(3):387–391, 1995.

[99] M. Püschel and B. Singer and M. Veloso and J. Moura. Fast Automatic Generation of DSP Algorithms. In *Proceedings of the International Conference on Computational Science, LNCS 2073*, pages 97–106, 2001.

[100] B.M. Maggs, L.R. Matheson, and R.E. Tarjan. Models of Parallel Computation: A Survey and Synthesis. In *Proceedings of the 28th Hawaii International Conference on System Sciences (HICSS)*, volume 2, pages 61-70, January 1995.

[101] P. Maurer. Logic Simulation Using Networks of State Machines. In *Proceedings of Design, Automation and Test in Europe Conference 2000 (DATE 2000)*, pages 674–678, Paris, France, March 2000.

[102] O.A. McBryan. An Overview of Message Passing Environments. *Parallel Computing*, 20(4):417–444, April 1994.

[103] W.F. McColl. Scalability, portability and predictability: The BSP approach to parallel programming. *Future Generation Computer Systems*, 12:265–272, 1996.

[104] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard (version 1.1). Technical report, University of Tennessee, Knoxville, Tennessee, June 1995. Available at `http://www.mpi-forum.org`.

[105] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface. Technical report, University of Tennessee, Knoxville, Tennessee, July 1997. Available at `http://www.mpi-forum.org`.

[106] D. Milicev and Z. Jovanovic. A Finite State Machine Based Formal Model of Software Pipelined Loops with Conditions. *International Journal of Computer Research*, 10(1):11–20, 2001.

[107] M. van der Molen and P. Jonker. A Comparison of Linear Processor Arrays for Image Processing. Technical report, Pattern Recognition Group, Faculty of Applied Sciences, Delft University of Technology, Delft, The Netherlands, 1998.

[108] M.S. Moore et al. A Model-Integrated Program Synthesis Environment for Parallel/Real-Time Image Processing. In *Parallel and Distributed Methods for Image Processing, Proceedings of SPIE*, volume 3166, pages 31–45, 1997.

[109] P.J. Morrow et al. Efficient Implementation of a Portable Parallel Programming Model for Image Processing. *Concurrency: Practice and Experience*, 11:671-685, 1999.

[110] T. Nakahara and T. Kanade. Experiments in Multi-Baseline Stereo. Technical report, Carnegie Mellon University, Computer Science Department, University, Pittsburgh, Pennsylvania, August 1992.

[111] C. Nicolescu and P. Jonker. EASY-PIPE - An Easy to Use Parallel Image Processing Environment Based on Algorithmic Skeletons. In *Proceedings of the 15th International Parallel & Distributed Processing Symposium - Workshop on Parallel and Distributed Computing in Image Processing, Video Processing, and Multimedia*. San Francisco, California, USA, April 2001.

[112] C. Nicolescu and P. Jonker. A Data and Task Parallel Image Processing Environment. *Parallel Computing*, 28(7–8):945–965, August 2002.

[113] D.S. Nikolopoulos et al. Is Data Distribution Necessary in OpenMP? In *Proceedings of the 2000 ACM/IEEE Supercomputing Conference on High Performance Networking and Computing (SC 2000)*, Dallas, Texas, USA, November 2000.

[114] N. Nupairoj and L.M. Ni. Performance Evaluation of Some MPI Implementations on Workstation Clusters. In *Proceedings of the 1994 Scalable Parallel Libraries Conference (SPLC94)*, pages 98–105, Mississippi State, Mississippi, USA, October 1994.

[115] N. Nupairoj and L.M. Ni. Performance Metrics and Measurement Techniques of Collective Communication Services. In *First International Workshop on Communication and Architectural Support for Network-Based Parallel Computing*, pages 212–226, San Antonio, Texas, USA, February 1997.

[116] S. Oberman, G. Favor, and F. Weber. AMD 3DNow! Technology: Architecture and Implementations. *IEEE Micro*, 19(2):37–48, March/April 1999.

[117] M. Okutomi and T. Kanade. A Multiple-Baseline Stereo. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(4):353–363, April 1993.

[118] J.G.E. Olk and P.P. Jonker. A Programming and Simulation Model of a SIMD-MIMD Architecture for Image Processing. In *Workshop on Computer Architecture for Machine Perception*, pages 98–105, September 1995.

[119] OpenMP Architecture Review Board. OpenMP C and C++ Application Program Interface. Technical report, `http://www.openmp.org`, October 1998.

[120] C.M. Pancake and D. Bergmark. Do Parallel Languages Respond to the Needs of Scientific Programmers? *IEEE Computer*, 23(12):13–23, December 1990.

[121] A. Peleg, S. Wilkie, and U. Weiser. Intel MMX for Multimedia PCs. *Communications of the ACM*, 40(1):24–38, January 1997.

[122] B.L. Peuto and L.J. Shustek. An Instruction Timing Model of CPU Performance. In *Proceedings of the Fourth Annual Symposium On Computer Architecture*, pages 165–178, March 1977.

[123] A. Pimentel. *A Computer Architecture Workbench*. PhD thesis, University of Amsterdam, The Netherlands, December 1998.

[124] M. Prieto, I.M. Llorente, and F. Tirado. A Review of Regular Domain Partitioning. *SIAM News*, 33(1), January 2000.

[125] M. Prieto, I.M. Llorente, and F. Tirado. Data Locality Exploitation in the Decomposition of Regular Domain Problems. *IEEE Transactions on Parallel and Distributed Systems*, 11(11):1141–1149, November 2000.

[126] S. Ramaswamy, S. Sapatnekar, and P. Banerjee. A Framework for Exploiting Task and Data Parallelism on Distributed Memory Multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 8(11):1098–1115, November 1997.

[127] N. Ranganathan. *VLSI and Parallel Computing for Pattern Recognition and Artificial Intelligence*. World Scientific Series in Machine Perception and Artificial Intelligence - Vol. 18, 1995.

[128] C. van Reeuwijk. Spar 1.0 Language Specification - Version Beta 1. Technical report, Delft University of Technology, Delft, The Netherlands, May 1999.

[129] C. van Reeuwijk, A.J.C. van Gemund, and H.J. Sips. Spar: A Programming Language for Semi-Automatic Compilation of Parallel Programs. *Concurrency: Practice and Experience*, 9(11):1193–1205, November 1997.

[130] A.K. Riemens et al. TriMedia CPU64 Application Domain Benchmark Suite. In *Proceedings of the International Conference on Computer Design*, Austin, Texas, USA, October 1999.

[131] G.X. Ritter and J.N. Wilson. *Handbook of Computer Vision Algorithms in Image Algebra*. CRC Press, Inc, 1996.

[132] R.H. Saavedra and A.J. Smith. Analysis of Benchmark Characteristics and Benchmark Performance Prediction. Technical Report USC-CS-92-524, Computer Science Department, University of Southern California, 1992.

[133] R.H. Saavedra-Barrera, A.J. Smith, and E. Miya. Machine Characterization Based on an Abstract High-Level Language Machine. *IEEE Transactions on Computers*, 38(12):1659–1679, December 1989.

[134] A. Saoudi and M. Nivat. Optimal Parallel Algorithms for Multidimensional Template Matching and Pattern Matching. In *Proceedings of the Second International Conference on Parallel Image Analysis, ICPIA '92*, pages 240–246, Ube, Japan, December 1992.

[135] C.H. Sauer and K. Mani Chandi. *Computer Systems Performance Modeling*. Prentice-Hall Series in Advances in Computing Science and Technology. Prentice-Hall, 1981.

[136] K. Schutte and G.M.P. van Kempen. Optimal Cache Usage for Separable Image Processing Algorithms on General Purpose Workstations. *Signal Processing*, 59(1):113–122, May 1997.

[137] F.J. Seinstra, H.E. Bal, and H.J.W. Spoelder. Parallel Simulation of Ion Recombination in Nonpolar Liquids. In *Proceedings of High-Performance Computing and Networking (HPCN'97)*, pages 213–222, Vienna, Austria, April 1997.

[138] F.J. Seinstra, H.E. Bal, and H.J.W. Spoelder. Parallel Simulation of Ion Recombination in Nonpolar Liquids. *Future Generation Computer Systems*, 13(4-5):261–268, March 1998.

[139] F.J. Seinstra and D. Koelma. Modeling Performance of Low Level Image Processing Routines on MIMD Computers. In *Proceedings of the Fifth Annual Conference of the Advanced School for Computing and Imaging*, pages 307–314, Heijen, The Netherlands, June 1999.

[140] F.J. Seinstra and D. Koelma. Transparent Parallel Image Processing by way of a Familiar Sequential API. In *Proceedings of the 15th International Conference on Pattern Recognition*, pages 824–827, Barcelona, Spain, September 2000.

[141] F.J. Seinstra and D. Koelma. The Lazy Programmer's Approach to Building a Parallel Image Processing Library. In *Proceedings of the 15th International Parallel & Distributed Processing Symposium - Workshop on Parallel and Distributed Computing in Image Processing, Video Processing, and Multimedia*, San Francisco, California, USA, April 2001.

[142] F.J. Seinstra and D. Koelma. Incorporating Memory Layout in the Modeling of Message Passing Programs. In *Proceedings of the 10th Euromicro Workshop on Parallel, Distributed and Network-Based Processing (PDP 2002)*, pages 293–300, Las Palmas de Gran Canaria, Canary Islands, Spain, January 2002.

[143] F.J. Seinstra and D. Koelma. P-3PC: A Point-to-Point Communication Model for Automatic and Optimal Decomposition of Regular Domain Problems. *IEEE Transactions on Parallel and Distributed Systems*, 13(7):758–768, July 2002.

[144] F.J. Seinstra and D. Koelma. Incorporating Memory Layout in the Modeling of Message Passing Programs. *Journal of Systems Architecture (in press)*, 2003.

[145] F.J. Seinstra and D. Koelma. Lazy Parallelization: A Finite State Machine Based Optimization Approach for Data Parallel Image Processing Applications. In *Proceedings of the 17th International Parallel & Distributed Processing Symposium - Workshop on Parallel and Distributed Computing in Image Processing, Video Processing, and Multimedia*, Nice, France, April 2003.

[146] F.J. Seinstra and D. Koelma. User Transparency: A Fully Sequential Programming Model for Efficient Data Parallel Image Processing. *Concurrency and Computation: Practice and Experience (in press)*, 2003.

[147] F.J. Seinstra, D. Koelma, and J.M. Geusebroek. A Software Architecture for User Transparent Parallel Image Processing on MIMD Computers. In *Proceedings of the 7th International Euro-Par Conference (Euro-Par 2001), Lecture Notes in Computer Science 2150*, pages 653–662, Manchester, UK, August 2001.

[148] F.J. Seinstra, D. Koelma, and J.M. Geusebroek. Bridging the Gap between Computing and Imaging: Towards 'Effortless' Parallel Image Processing. In *Proceedings of the Seventh Annual Conference of the Advanced School for Computing and Imaging*, pages 443–450, Heijen, The Netherlands, May 2001.

[149] F.J. Seinstra, D. Koelma, and J.M. Geusebroek. A Software Architecture for User Transparent Parallel Image Processing. *Parallel Computing*, 28(7–8):967–993, August 2002.

[150] F.J. Seinstra, D. Koelma, J.M. Geusebroek, F.C. Verster, and A.W.M. Smeulders. Efficient Applications in User Transparent Parallel Image Processing. In *Proceedings of the 16th International Parallel & Distributed Processing Symposium - Workshop on Parallel and Distributed Computing in Image Processing, Video Processing, and Multimedia*, Fort Lauderdale, Florida, USA, April 2002.

[151] H.J. Siegel et al. Mapping Computer Vision-Related Tasks onto Reconfigurable Parallel-Processing Systems. *IEEE Computer*, 25(2):54–63, February 1992.

[152] B. Singer and M. Veloso. Learning to Generate Fast Signal Processing Implementations. In *Proceedings of the Eighteenth International Conference on Machine Learning*, pages 529–536, 2001.

[153] J.M. Squyres et al. Cluster-Based Parallel Image Processing. Technical report, Laboratory for Scientific Computing, Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, Indiana, USA (TR 96-9), 1996.

[154] J.M. Squyres, A. Lumsdaine, and R.L. Stevenson. A Toolkit for Parallel Image Processing. In *Parallel and Distributed Methods for Image Processing II, Proceedings of SPIE*, San Diego, California, USA, July 1998.

[155] J.A. Steele. *An Abstract Machine Approach to Environments for Image Interpretation on Transputers.* PhD thesis, Faculty of Science, Department of Computer Science, The Queen's University of Belfast, N. Ireland, May 1994.

[156] A.J. van der Steen. Is it Really Possible to Benchmark a Supercomputer? A graded approach to performance measurement. In A.J. van der Steen, editor, *Evaluating Supercomputers : strategies for exploiting, evaluating and benchmarking computers with advanced architectures*, chapter 14, pages 190–212. Chapman and Hall, 1990.

[157] T. Sterling et al. BEOWULF: A Parallel Workstation for Scientific Computation. In *Proceedings of the 24th International Conference on Parallel Processing*, pages I:11–14, Oconomowoc, Wisconsin, USA, August 1995.

[158] B. Stroustrup. *The C++ Programming Language, 3rd Edition.* Addison-Wesley, 1997.

[159] R. Taniguchi et al. Software Platform for Parallel Image Processing and Computer Vision. In *Parallel and Distributed Methods for Image Processing, Proceedings of SPIE*, volume 3166, pages 2–10, 1997.

[160] J.J. Temminck. *Haarlemmer Halletjes - Grepen uit de Geschiedenis van Haarlem en Omgeving.* Excelsior Haarlem B.V., 2000.

[161] S.H. Unger. A Computer Oriented towards Spatial Problems. *Proceedings of the Institute of Radio Engineers*, 46:1744–1750, 1958.

[162] L.G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[163] A.J. van der Steen and R. van der Pas. A Performance Analysis of the SGI Origin 2000. In *Proceedings of the Third International Meeting on Vector and Parallel Processing*, pages 534–547, Porto, Portugal, June 1998.

[164] R.S. Wallace, J.A. Webb, and I.C. Wu. Machine-Independent Image Processing: Performancy of Apply On Diverse Architectures. *Computer Vision, Graphics and Image Processing*, 48(2):265–276, 1989.

[165] J.A. Webb. Steps Toward Architecture-Independent Image Processing. *IEEE Computer*, 25(2):21–31, February 1992.

[166] J.A. Webb. Implementation and Performance of Fast Parallel Multi-Baseline Stereo Vision. In *Proceedings of the 1993 DARPA Image Understanding Workshop*, pages 1005–1010, April 1993.

[167] R.P. Weicker. Dhrystone: A Synthetic Systems Programming Benchmark. *Communications of the ACM*, 27(10):1013–1030, October 1984.

[168] F. Weil, L.H. Jamieson, and E.J. Delp. Dynamic Intelligent Scheduling and Control of Reconfigurable Parallel Architectures for Computer Vision/Image Processing. *Journal of Parallel and Distributed Computing*, 13:273–285, 1991.

[169] R.C. Whaley and J.J. Dongarra. Automated Empirical Optimization of Software and the ATLAS Project. *Parallel Computing*, 27(1–2):3–25, 2001.

[170] G.V. Wilson. *Practical Parallel Programming.* Scientific and Engineering Computation Series. The MIT Press, 1995.

[171] G.V. Wilson and P. Lu. *Parallel Programming Using C++.* Scientific and Engineering Computation Series. The MIT Press, 1996.

[172] Z. Xu, X. Zhang, and L. Sun. Semi-Empirical Multiprocessor Performance Predictions. *Journal of Parallel and Distributed Computing*, 39(1):14–28, 1996.

[173] X. Zhang, Y. Yan, and K. He. Latency Metric: An Experimental Method for Measuring and Evaluating Parallel Program and Architecture Scalability. *Journal of Parallel and Distributed Computing*, 22(3):392–410, 1994.

Samenvatting*

"Alle intellectuele beroepen bestaan uit het continu verrichten van dingen die, apart genomen, heel eenvoudig zijn."

Willem Frederik Hermans - *Nooit Meer Slapen* (1966)

Om te kunnen voldoen aan de specifieke prestatie-eisen van huidige en toekomstige beeldverwerkingsapplicaties, toont de beeldverwerkingsgemeenschap een grote wens de rekenkracht van parallelle en gedistribueerde computersystemen aan te wenden. Helaas bestaat er vooralsnog een enorme discrepantie tussen de vraag naar *eenvoudig te verkrijgen* rekenkracht, en de wijze waarop de aanwezige rekenkracht ook daadwerkelijk wordt aangeboden. De kern van dit probleem bestaat eruit, dat voor beeldverwerkingsonderzoekers geen effectief en laagdrempelig hulpmiddel voor handen is voor het ontwikkelen van parallelle applicaties. Bestaande hulpmiddelen vereisen van de gebruiker een inzicht in de complexiteit van parallellisme dat verder gaat dan van beginnende parallelle programmeurs verwacht mag worden. Omdat het niet reëel is te eisen dat experts op het gebied van beeldverwerking tevens experts op het gebied van parallel programmeren worden, is het noodzakelijk dat een hulpmiddel wordt ontwikkeld dat gebruikers afschermt van *alle* parallellisatieproblematiek.

Het in dit proefschrift beschreven onderzoek poogt de kloof te dichten tussen de specifieke expertise van beeldverwerkingsonderzoekers, en de additionele kennis die vereist is voor het efficiënt aanwenden van parallelle computers. Daartoe behandelt het proefschrift het ontwerp en de implementatie van een softwarearchitectuur die beeldverwerkingsonderzoekers in staat stelt parallelle applicaties te ontwikkelen *op een voor de gebruiker volledig transparante wijze* (dat wil zeggen: volledig sequentieel). Het proefschrift onderzoekt de specifieke eisen die aan een dergelijke architectuur gesteld moeten worden, zodat het dienst kan doen als een voor beeldverwerkers acceptabele langetermijnoplossing. Daarnaast geeft het proefschrift een gedetailleerde verhandeling van de verschillende componenten van de ontwikkelde architectuur, alsmede van de daaraan gerelateerde onderzoeksvragen. Eveneens geeft het proefschrift een uitgebreide evaluatie van de door de architectuur geleverde snelheidswinsten, gecombineerd met een vergelijking met handgeoptimaliseerde applicaties.

*Summary in Dutch

Hoofdstuk 2 onderzoekt de mate waarin bestaande hardwarearchitecturen, en daaraan gerelateerde programmeerhulpmiddelen, geschikt zijn voor toepasssing in beeldverwerkingsonderzoek. Aan de hand van een lijst van specifieke eisen wordt geconcludeerd dat de klasse van Beowulf-clusters het meest geschikt is, voornamelijk vanwege de positieve prijs-prestatieverhouding. De evaluatie van programmeerhulpmiddelen toont aan dat bibliotheekgebaseerde architecturen een oplossing bieden die het meest aansluit bij de specifieke wensen van de beeldverwerkingsgemeenschap. Dit komt voornamelijk doordat dergelijke architecturen op een redelijk eenvoudige wijze geleverd kunnen worden met een programmeermodel dat de gebruiker volledige transparantie biedt. Echter, vanwege gebrekkige *onderhoudbaarheid* bestaat er op dit moment geen architectuur die ook daadwerkelijk beschouwd kan worden als een acceptabele langetermijnoplossing. Op basis van deze observaties wordt een nieuwe bibliotheekgebaseerde architectuur voor parallel beeldverwerking geïntroduceerd. Door het innovatieve ontwerp voldoet de architectuur zowel aan de eis van onderhoudbaarheid, als aan de eis van gebruikerstransparantie.

Hoofdstuk 3 beschrijft de ontwerpfilosofie van de bibliotheek, die de kern vormt van de ontwikkelde architectuur. Centraal staat de vraag hoe de bibliotheek zodanig geïmplementeerd kan worden dat coderedundantie zoveel mogelijk wordt vermeden, met behoud van efficiëntie. Daartoe wordt het begrip *parallelliseerbare patronen* geïntroduceerd, en getoond hoe parallelle implementaties eenvoudig verkregen kunnen worden door concatenatie van operaties die alle *apart* aanwezig zijn in de bibliotheek. Op basis van vier verschillende typen van gegevensbenadering wordt een standaard parallellisatiestrategie gedefinieerd voor elke operatie die overeenkomt met een van de parallelliseerbare patronen. De standaard parallellisatiestrategie is te allen tijde optimaal, omdat het al het aanwezige parallellisme volledig benut met een minimum aan communicatiekosten. De kenmerken van de gepresenteerde ontwerpfilosofie verzekeren langetermijnonderhoudbaarheid van de architectuur, alsook efficiëntie.

Hoofdstuk 4 geeft aan hoe een eenvoudig analytisch prestatiemodel toegepast wordt in het proces van automatische parallellisatie en optimalisatie van beeldverwerkingsapplicaties. Bestaande modellen bevatten in het algemeen een directe complexiteit-nauwkeurigheidrelatie, en zijn daarom voor de gewenste accuratesse niet efficiënt genoeg. Om dit probleem het hoofd te bieden wordt een semi-empirische modelleertechniek geïntroduceerd, die naast eenvoud en overdraagbaarheid tevens voldoende nauwkeurigheid biedt voor de gegeven doelstelling. De techniek is gebaseerd op de definitie van een abstracte machine (*APIPM*), die de typische gedragingen van parallelle beeldverwerkingsoperaties beschrijft. Op basis van de gerelateerde APIPM-instructieset wordt een prestatiemodel verkregen dat algemeen toepasbaar is op Beowulf-clusters. De essentie van de modelleertechniek bestaat eruit, dat van bepalende kostenfactoren die Beowulf-clusters niet alle gemeen hebben geabstraheerd wordt in de definitie van de modelparameters. Om elke abstracte modelparameter alsnog te verbinden met een prestatieschatting voor een concrete parallelle machine, wordt een klein aantal tijdmetingen uitgevoerd die alle impliciete, doch essentiële, kostenfactoren ondervangen. Een vergelijking van gegenereerde schattingen met experimentele resultaten toont de hoge nauwkeurigheid van de APIPM-gebaseerde modellen voor realistische applicaties.

Hoofdstuk 5 breidt de APIPM-gebaseerde modellen uit om nauwkeuriger schattingen te verkrijgen voor de MPI-primitieven die in de bibliotheek worden toegepast. Bestaande communicatiemodellen (zoals LogP) ondervangen niet alle mogelijke gedragingen van de MPI-operaties. Tot nu toe is het (vaak zeer significante) effect van de schikking van gegevens in het geheugen op de communicatiekosten volledig genegeerd. In de ontwikkelde architectuur is een hogere nauwkeurigheid essentieel om de automatische distributie van beeldgegevensstructuren optimaal uit te voeren. Daarom wordt het nieuwe P-3PC-model geïntroduceerd, dat het typische gedrag van de MPI-primitieven juist representeert. Ten eerste ondervangt het de verschillen in communicatiekosten voor de zender, de ontvanger, alsook het volledige communicatiepad. Daarnaast modelleert het het effect van geheugenschikking, en ondervangt het kosten die niet-lineair afhankelijk zijn van de berichtgrootte. Experimenten die zijn uitgevoerd op twee verschillende Beowulf-clusters tonen aan dat, in vergelijking met gerelateerde modellen, P-3PC in staat is nauwkeuriger schattingen te leveren voor de communicatiekosten zoals die bestaan in typische beeldverwerkingsapplicaties.

Hoofdstuk 6 behandelt de automatische conversie van elke sequentiële beeldverwerkingsapplicatie naar een correcte en efficiënte parallelle versie. Daartoe wordt een eindige automaat gedefinieerd, die garandeert dat dit proces te allen tijde correct wordt uitgevoerd. Ten eerste wordt aangetoond dat de automaat leidt tot een opvallend eenvoudige methode voor minimalisatie van de communicatiekosten. Voor verdere prestatieoptimalisaties wordt de automaat toegepast in de constructie van een graaf, die alle mogelijke toestandsveranderingen en parallellisatiestrategieën van een draaiende applicatie weergeeft. Omdat voor elke mogelijke strategie een APIPM-gebaseerde kostenschatting voor handen is, wordt de snelste versie van een applicatie weergegeven door het goedkoopste pad in de graaf. Vanwege het feit dat de automatische optimalisatie van volledige applicaties het centrale, meest essentiële, probleem vormt waarmee de ontwikkelde softwarearchitectuur wordt geconfronteerd, combineert de toegepaste oplossing alle resultaten zoals bereikt in de Hoofstukken 3, 4, en 5.

Hoofdstuk 7 geeft een inschatting van de effectiviteit van de softwarearchitectuur in het behalen van significante prestatieverbeteringen. Daartoe wordt een beschrijving gegeven van de sequentiële implementatie en automatische parallellisatie van drie voorbeeldapplicaties, die alle veelvoorkomende beeldverwerkingsoperaties bevatten. Op basis van de evaluatie wordt geconcludeerd dat de snelheidswinsten die behaald worden door de parallelle applicaties zoals gegenereerd door de ontwikkelde softwarearchitectuur vergelijkbaar zijn met handgeoptimaliseerde implementaties.

Conclusie: de resultaten bereikt met de ontwikkelde softwarearchitectuur tonen aan dat het mogelijk is een oplossing te ontwikkelen voor een ingewikkeld probleem, voornamelijk door het combineren van relevante abstracties, en toepassing van relatief eenvoudige methoden voor de oplossing van elk deelprobleem. Bijgevolg lijkt het er sterk op dat een vergelijkbare strategie toepasbaar is op andere onderzoeksterreinen, voornamelijk daar waar het aantal typische operaties gelimiteerd is — net zoals dat het geval is in beeldverwerkingsonderzoek. Desalniettemin kan alleen de tijd leren of de ontwikkelde softwarearchitectuur ook daadwerkelijk de door ons vermoedde effectieve langetermijnoplossing zal zijn.

Acknowledgements

"Parallel lives — running parallel with you
To the point where our horizons divide..."

Fates Warning - *Parallels* (1991)

Pursuing a Ph.D. within a research group whose main focus differs quite substantially from that of the work described in this thesis has not always been an easy task. As a result, in cooperating with other group members it was important for me to always remember the main motto of this thesis: "Mind the gap!". Nonetheless, my years within ISIS have been a tremendous experience, scientifically as well as personally.

For giving me the opportunity to do my research, and for letting me have all freedom, I am principally grateful to my promotor, Arnold Smeulders. After I had wandered off in a direction that did not suit me, he kept his confidence in my capabilities, and allowed me to do what I liked most: parallel computing. Also, in the organization of 'het AiO-overleg' he showed great cooperation, understanding, and leadership. But primarily I would like to thank you, Arnold, for your never diminishing urge to challenge my reluctance to learn things of which I didn't see the essence immediately.

Dennis, although I know that I am supposed to say the nicest of things about you on these pages, I simply have no alternative. Your style of supervision ("do whatever you like, as long as it is at least remotely useful") was exactly what suited me best. Because of this laid-back attitude, and also because of your unmatched competence in software engineering, I am very much looking forward to continue working together with you in the near future. The first and foremost thing on our 'to-do list', however, should be nothing other than an adequate bottle of single malt Scotch!

The regular meetings with 'the Delft connection' (i.e., Cristina Soviany, Henk Sips, and Pieter Jonker) provided me with a wealth of information and insight. Although our cooperation sometimes was a 'parallelization problem' in itself, this thesis would not have been the same without this input. I will never forget Pieter's enthusiasm for the project, or his three-hour crash course in parallel architectures. I am also grateful for Henk's never-ending urge to make me think more carefully about the concepts and theory behind my chaotic 'hersenspinsels'. In addition, in the future I will definitely take heed of Cristina's results on combined data and task parallelism.

Many thanks go out to all of my (past and present) ISIS group members as well. In particular, I would like to thank Jan-Mark Geusebroek for providing me with the coolest of example applications, for taking the time to try to understand the potential of my software from as early as 1999, and for realizing that shouting 'Biertje?' at the most ridiculous of times was exactly what I needed in the process of writing this thesis. I would also like to thank Edo Poll for our discussions on engineering problems, for providing (part of) the basis for the APIPM abstractions, and simply for his great achievements in the Horus project. In addition, I would like to thank Silvia Olabarriaga and Wilko Quak for our legendary 'idiot nights', and Leon Todoran, Carlo de Boer, and Jeroen Vendrig for taking part in one of the greatest events in my life: the solar eclipse in Romania in 1999. Many thanks also to Andrew Bagdanov and Joost van de Weijer, for their enthusiasm and interest in a great diversity of topics — ranging from scientific, to cultural, political, and more 'potable' issues. Special appreciation goes out to Virginie Mes, for being the greatest, reliable, and friendly source of help that she is, especially in situations of complete panic. I would also like to say sorry to Saskia Heijboer and Hette Knol for annoying them with anything 'important' in case Virginie was not around. Also, it was a great pleasure to have met several non-ISIS members whom I have learned to know just a little bit better along the way, especially: Anne Frenkel, Cesar Garita, Victor Guevara Masis, Joris Portegies Zwart, and Elena Zudilova. Finally, I would like to thank Thang Pham, for being the finest room mate I could wish for. Thang, please forgive me for being moody every now and then. I promise that I will not write a thesis ever again!

I consider myself lucky to have many great friends who have made life bearable the last few years. First, I would like to thank Hans de Wit, for being the great person that he is. Now that I've got some more time, I am definitely looking forward to visit new 'strategic' cities with you in the future! Next, I would like to give a big hug to Jacqueline van der Velde. Despite the fact that our horizons have divided somewhat, you are still a great source of inspiration — and of new movies, of course. I would also like to thank Dennis Kool, for our discussions on all aspects of life (esp. the feminine ones), and for letting me win any game of eight-ball I play against you (hehe). Special thanks go out to Edwin 'Het Orakel van Alanya' Steensma for driving me all around The Netherlands on our '4-daagse' training days, for being a great source of classical and metal music, as well as British real ales, and simply for being the craziest (and probably loudest) person the world has ever seen. In no particular order I would also like to say 'hello' to Micha van der Velden, Laura de Vries, Martijn van der Velden, Erik Stel, Pieter de Wit, Marco Essers, Roland Schmidt, Marco Heijkoop, Folkert 'Zonder V' Jongsma, Simon-Jan Smit, Sverda Miedema, and all others I forgot.

Finally, special gratitude goes out to my mother. A cliche, but true nonetheless: this thesis would never have existed without her. Throughout the years she has been the greatest source of motivation, support, and above all: understanding. Whenever life showed an ugly face, she was there to listen, and to keep me focussed on what I often held as the least probable achievement in the world. Maar... Mam, het is gelukt!

Now, if you all allow me, I would like to retreat for an hour or so. You can bet I'll be enjoying Sibelius' 2nd Symphony, together with a nice glass of Aventinus Dunkelweizen... And I will be perfectly happy... - Frank.

Stellingen

behorend bij het proefschrift
"User Transparent Parallel Image Processing"
door Frank J. Seinstra

1. An automatic parallelization tool constitutes a long-term solution for the image processing research community only, if it is sustainable and fully user transparent. *(chapter 2 of this thesis)*

2. *Parallelizable patterns* are the design strategy of choice for lazy parallel programmers. *(chapter 3 of this thesis)*

3. Application of *semi-empirical modeling* in a library-based software architecture removes the need for auxiliary benchmarking kernels. *(chapter 4 of this thesis)*

4. The effect of memory layout on communication costs often is more substantial than the effect of network contention. *(chapter 5 of this thesis)*

5. For a library-based parallelization tool to obtain competitive performance, optimization across library calls is essential. *(chapter 6 of this thesis)*

6. Even in case of full user transparency, part of the efficiency of parallel execution inherently remains in the hands of the application programmer.
(chapter 7 of this thesis)

---

7. Het moeilijkste van het schrijven van een proefschrift is het aan buitenstaanders uitleggen wat dat precies inhoudt.

8. Het toevoegen van een motto aan elk hoofdstuk heeft als effect dat de meerderheid van de lezers een langere tijd uittrekt voor het doornemen van het proefschrift; immers, de tijd die de modale lezer spendeert aan het bekijken van de stellingen en het dankwoord wordt nu uitgebreid met de tijd die besteed wordt aan het lezen van de motto's.

9. Professoren drukken zich vaak slecht uit, maar AiO's moeten daar beter naar luisteren.
(vrij naar het leermoment "Mannen drukken zich slecht uit, maar vrouwen moeten daar beter naar luisteren" uit de RVU-serie "Mannen voor Vrouwen".)

10. Door 'publish or perish' komt uiteindelijk *iedereen* om in het papier.

11. Het idealiseren van succes gaat voorbij aan het feit dat de essentie van ieders individu mede wordt bepaald door hetgeen men *niet* bereikt heeft, en door dat wat men is kwijtgeraakt.

ISBN 90-5776-102-5